APPLE
組合語言

陸文麟 譯



II

協群科技出版社

果粉藏書

APPLE 組合語言

陸文麟 譯

0005520

712

協群科技出版社

APPLE組合語言

編譯者：陸　文　麟

出　版：協群科技出版社

發　行：協群科技出版社

香港中環皐利街684號二樓

印刷者：廣　源　印　務　局

青山道875號工廠大厦

定價：H.K.$ 25.00

前　　言

組合語言（Assembly Language）叫人害怕的唯一原因，可能就是它怪怪的，有點工業化味道的名字了；那末，爲什麼不把它當成一套〝友善的語言〞來看呢？因爲它正是爲您而設的啊！這本書正是爲那些需要使用並且玩賞他的Apple的普羅大衆而寫的，所以在寫作上就採取了能夠讓初學者容易了解、容易跟得上的法子。隨著這本書淸晰而且漸進的方式，帶您講覽這套語言，相信您會覺得有很多人感到〝組合語言不好學〞這件事有點値得懷疑了。

您還可以唬您的朋友，用組合語言寫出來的程式要比用BASIC做的有時候會快上100倍!圖形、卡通、電動玩具遊戲，還有很多其它的應用，用組合語言來寫，是更爲生動的；還有，了解一點組合語言也正是開啓Apple的監督程式、DOS、與其它系統程式的大門，也爲您要充分了解Apple舖了一條坦途。

您曉得嗎？在Apple ROM的深處有一個神秘兮兮的，叫做Sweet—16的微電腦!!!!這本書還教您如何用它來減少您的組合語言程式指令數目；這可是在其它的初等教材上找不到的哦………。

Retro Workshop
果粉工作室

目 錄

第一章　導　論 …… 1

手册的目的 …… 1

手册的範圍 …… 1

通　論 …… 1

為何要用組合語言 …… 5

第二章　符　號 …… 6

概　論 …… 6

位元字串 …… 9

二元算術 …… 14

不帶符號的整數 …… 16

尼波，位元組和字 …… 17

帶符號的整數 …… 18

十六進位數目 …… 20

底數和其毛病 …… 21

ASCII字元集合 …… 22

用字元串來表示指令 …… 23

第三章　暫時器、指令型式和位址 …… 25

概　論 …… 25

累積器 …… 27

X 暫存器 …… 27
Y 暫存器 …… 28
堆疊指標 …… 28
程式狀況字 …… 28
程式標示 …… 28
指令型式 …… 29
二位元組和三位元組指令 …… 30
6502 的位址型式 …… 32
直接位址型式 …… 33
絕對位址型式 …… 33
零頁位址型式 …… 34
索引位址型式 …… 34
間接位址 …… 35
由 Y 間接索引 …… 35
間接的由 X 索引 …… 35
暗示的位址型式 …… 35
累積器位址型式 …… 36
相對位址型式 …… 36
結　語 …… 36

第四章　一些簡單指令 …… 37

新指令： …… 37
概　論 …… 37
組合語言的程式型式 …… 38
有效的標籤例子 …… 38
符號欄 …… 39
運算元欄 …… 39

註解欄 …… 40
載入群 …… 40
存入指令 …… 42
資料轉移指令 …… 43
暫存器的增加和減少 …… 44
增加和減少指令 …… 45
標示和變數 …… 45
運算元欄中的例子 …… 48

第五章 組合語言 …… 49

新指令 …… 49
概　論 …… 50
示範程式 …… 50
跳開指令 …… 52
處理機狀況暫存器 …… 54
BREAK 旗標(B) …… 54
十進位旗標 (D) …… 54
廢除中斷旗標 …… 55
情況碼旗標 …… 55
轉移指令 …… 57
迴　路 …… 58
比　較 …… 60
IF/THEN 陳述的模擬 …… 62
FOR/NEXT迴路 …… 62
布林值的測試 …… 66

第六章 算術運算 …… 69

新指令 …… 69

概　論……………………………………………… 69
不帶符號整數(二元數)算術…………………… 69
不帶符號加法的規則……………………………… 72
減法………………………………………………… 72
不帶符號減法的規則……………………………… 73
帶符號算術………………………………………… 73
帶符號算術的規則………………………………… 75
帶符號的比較……………………………………… 75
二元碼十進位算術………………………………… 76
不帶符號的BCD算術……………………………… 77
十進位算術範例…………………………………… 78
不帶符號算術的規則……………………………… 78
帶符號的BCD算術………………………………… 79
摘　要……………………………………………… 79
8位元算術的規則………………………………… 80

第七章　副程式和堆疊處理……………………… 81

新指令……………………………………………… 81
概　論……………………………………………… 81
變數問題…………………………………………… 83
傳送參數…………………………………………… 93

第八章　陣列第零頁索引和間接位址法………… 95

新指令……………………………………………… 95
概　論……………………………………………… 95
第零頁位址法……………………………………… 95
組合語言中的陣列………………………………… 97
在編譯時給定陣列初值……………………………102

用索引位址法來處理陣列元素 ……… 104
間接位址法 ……… 106
間接索引位址法 ……… 110
索引間接位址法 ……… 111
第九章　邏輯罩幕和位元運算 ……… 113
新指令 ……… 113
概　論 ……… 113
補數函數 ……… 114
AND 函數 ……… 114
OR 函數 ……… 115
XOR 函數 ……… 116
位元字串運算 ……… 117
邏輯運算的指令 ……… 118
AND 指令 ……… 118
ORA 指令 ……… 119
XOR / EOR指令 ……… 120
取累積器的補數 ……… 120
罩幕運算 ……… 121
罩　除 ……… 121
罩　進 ……… 126
移動和旋轉指令 ……… 127
算術往左移（ASL ）指令 ……… 127
邏輯往右（LSR ）指令 ……… 129
往左旋轉（ ROL）指令 ……… 130
往右旋轉（ ROR）指令 ……… 131
移動和旋轉記憶體位置 ……… 131

用 ASL 來作乘法……132
用移動來聚集資料……134
用旋轉和移動來聚集資料……135
第十章 多重精確度運算……137
概 論……137
多重精確度的邏輯運算……137
多重精確度的移動和旋轉……139
多重精確度的邏輯往右移動系列……140
多重精確度的往左旋轉……141
多重精確度的往右旋轉……142
多重精確度的不帶符號算術……143
N位元組不帶符號的加法規則……144
多重精確度的無號減法……145
二位元組的減法範例……145
多重精確度減法的規則……145
多重精確度的有號算術……146
多重精確度的十進位算術……146
多重精確度的增加……146
多重精確度的減指令……147
多重精確度的不帶符號比較……148
測試——16 位元值是否爲 0……148
測試——16 位元值是否爲負數……149
相等和不等的測試……149
帶符號的比較……152
第十一章 基本的輸出輸入……155
概 論……155

字元輸出 …… 155
標準輸出和週邊裝置 …… 163
字元輸入 …… 165
一行字元的輸入 …… 168

第十二章　數值的I/O …… 171

概　論 …… 171
16　進位的輸出 …… 171
十進位值的位元組輸出 …… 173
16　位元不帶符號整數的輸出 …… 175
帶符號16位元整數的輸出 …… 176
輸出整數的一種簡單方法 …… 177
數值輸入 …… 178
16 進位和BCD …… 178
不帶符號十進位的輸入 …… 181
帶符號十進位的輸入 …… 187

第十三章　乘法和除法 …… 191

概　論 …… 191
乘　法 …… 191
除法演算法 …… 191

第十四章　處理字串的運算 …… 203

字串的處理 …… 203
宣告字串常數 …… 207
字串設定 …… 207
字串函數 …… 209

字串串連……………………………………………211
次字串運算…………………………………………213
字串的比較…………………………………………215
字元陣列的處理……………………………………220

第十五章　APPLE Ⅱ 特殊的輸入方式……………225

APPLE 輸出入結構…………………………………225

第十六章　SWEET-16 簡介…………………………235

SWEET-16 ……………………………………………235
SWEET-16 硬體的要求………………………………246

第十七章　程式的偵錯與除錯………………………247

GO-指令(G)…………………………………………248
暫存器與記憶體的起始值……………………………249
修正指令碼（補正）…………………………………253
程式偵錯例子…………………………………………257

附　錄 ……………………………………………………261

第一章
導 論

手册的目的（*PURPOSE OF MANUAL*）

這本手册提供了 APPLE II 機器可用的6502組合語言的指令。所包含的內容適用於初級，中級和高級程式設計師。

手册的範圍（*SCOPE OF MANUAL*）

書中包括了基本符號和常用術語的解釋。也包括計算機概念的介紹，簡單的組合語言例子及與 APPLE II 有關的6502組合語言指令的介紹。

通 論（*GENERAL*）

爲什麼要另外寫一本專講6502 組合語言的書呢？原因是第一，目前這方面的書只有二本。第二，其中沒有一本是專門針對 APPLE II 計算機而討論的。雖然你可從這些書中學到組合語言的理論，可是對擁有 APPLE II 的使用者而言

，書中的範例卻一點用也沒有。

這本書是我做為6502 組合語言指導時所得經驗的累積。書中的內容對初學者而言並不難。雖然不能保證看完此書能精通到何種程度，但可以確定的是你一定可以成爲具有中等程度的6502 組合語言的程式員，“精通”的階段仍是需要經過幾年的經驗才可達到的。

假如你曾用過6502 ，則頭幾章所講的你可能已經知道了。但不要跳過任何一段！某個重要的細節若不詳細了解，則會影響到整本書其他部分的了解程度。因此要看過書中所有的資料且在繼續下去之前要確信你已了解了複習的部分。假如你是初學者，則一定要了解每一節之後才能繼續看下去!

有很多關於計算機理論和微計算機的書，因此我把關於這些的討論儘量減少。假如你對6502 組合語言很有興趣下列的書我推薦你應該去買來看：

HOW TO PROGRAM MICROCOMPUTERS
by William Barden Jr.

PROGRAMMING THE 6502
by Rodney Zaks

PROGRAMMING A MICROCOMPUTER
by Caxton C. Foster

6502 ASSEMBLY LANGUAGE PROGRAMMING
by Lance Leventhal

6502 SOFTWARE GOURMET GUIDE & COOKBOOK
by Robert Findley

雖然前面幾本書都很好，可是都不是與 APPLE II 直接相關的。假如你想學好組合語言則你該看過前面所提的幾本書，和這本手册。

在討論組合語言之前，先讓你熟悉一些以後經常會常用到的名詞：

RAM （隨機處理記憶體）：使用者可以使用的記憶体。程式和資料都存在RAM中。

ROM （僅讀記憶體）：用來放APPLE的監督程式（monitor ）和BASIC的地方。使用者不可將程式或資料存在此處。

MONITOR（監督程式：僅讀記憶體中的一組副程式，可以使你讀入鍵盤輸入的資料，在銀幕上顯示出字元等。

BASIC：代表整數BASIC。

K：當看到K時，就把它代換成“ × 1024 ”（即乘以1024 ），通常用來表示記憶體的大小。（如48K）。

記憶體（ memory ）：爲所有RAM和ROM的組合。

帶符號數字：任何合法的正整數或負整數（在目前運算下合法的數字）。

不帶符號數字：任何合法的正整數。不允許使用負整數。

位元組：記憶體的一種單位。一個位元組可代表256種不同的數值（如0-255之間的整數）。

字（ WORD ）：由二個位元組連成的。用一個“字”，可代表65,536種不同的數值（如0～65535之間的整數，或－32768～32767之間的帶符號數字）。

文法（ syntax ）：程式語言中用來規定句子結構的法則

位址（ address ）：爲二個位元組所組成的，用來指到64K個可用記憶位置中的某一個。一個位址也是一

個字，但一個字不一定是一個位址。

頁（PAGE）：APPLE Ⅱ 計算機中65536個位元組被分成256個片段，每個片段由256個位元組組成。每個片段稱爲一頁，編號從0到255。

第0頁：記憶體中最前面的256個位元組稱爲第0頁。當然也有第一頁、第二頁，可是在機器中最常用的是這一個片段，因此特稱爲第0頁”。

擴充接點（solt）：APPLE Ⅱ 計算機中具有8個與週邊設備連接的擴充接點。

I/O：輸入/輸出。

LISA：爲 Lazer System Interactive Symbolic Assembler（雷射系統接觸式組合編譯器）的簡稱。

週邊裝置（peripheral）：與計算機相連接的外部輸出入裝置。

假設你已熟悉 Apple 的 BASIC 語言。本書中只在一些例子中會用到 BASIC，但如你已熟悉 BASIC 那表示你已具有基本的程式設計技巧了。你應先了解一些程式技巧之後再來學組合語言。組合語言牽涉較廣，如果你一邊學最基本的程式技巧一邊學組合語言，則很容易會使你自己處於混亂的狀況下。

學習任何程式語言，尤其是組合語言，都要有實際操作的經驗。書中的例子都用 LISA（爲 APPLE II 的6502組合編譯器）。LISA 對初學者最適合，因爲它是接觸式的，也就是當每一行程式進入機器後，系統會立刻抓出文法上的錯誤。這與 APPLE II 中的整數 BASIC 很像。其餘 APPLE II Ⅱ 可用的組合編譯器都沒這樣的功能。

爲何要用組合語言
(*WHY USE ASSEMBLY LANGUAGE?*)

當速度是程式中首要的要求時，或是當你需要控制某種週邊裝置時或是利用高階語言無法完成你的應用問題時，就要使用組合語言。

不要把組合語言用在商業或科學用途上，因爲在這方面的工作中， Pascal, FORTRAN 或 Applesoft 會更適用。浮點的運算雖然不是不可爲，但可能太難，不是初學者或中級程式員能夠處理的。

組合語言所提供的另一個好處是可與現有的 BASIC 或 Applesoft 或 Pascal 的程式互相連接。你可以用組合語言寫時間要求較嚴格的部分，而其他部分則以 BASIC 來寫。

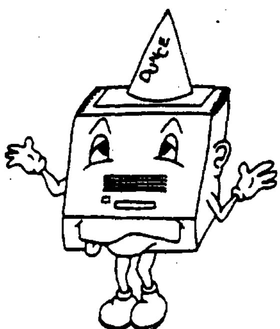
一旦你對 組合語言熟悉了之後，你會發現它的編寫和除錯與 BASIC 程式的編寫和除錯一樣地簡單！

Retro Workshop 果粉工作室

第二章
符 號

概 論（GENERAL）

當你看到數字“ 4 ”時，你想到什麼？數字“ 4 ”只是用來表示4 個項目這個觀念的符號。人們在溝通時，利用許多符號來表達意思。同時，人們的適應力也相當強，假如我說從現在開始，用“ —— ”來代表4 ，你一定可以改得過來。雖然不一定簡單，卻一定可行。



COMPUTERS ARE DUMB!

計算機卻是很笨的東西，它無法改變而且只能了解一種層次很低而人們認爲很難了解的語言，但這並不是組合語言或機器語言。組合或機器語言是人們用來使更低階的語言較

易被接受的工具。計算機可以了解的語言是由機器中許多電線上不同的伏特值所組成的。雖然，在受過許多訓練後，人們可以了解這些伏特值的意義，可是畢竟並不方便。通常我們把伏特值用其他方式代表（位元、眞假、0，1等）。這種情形在平常語言中也有相同的情況，如把法文中的deux翻成英文中的" two "。把伏特值改稱爲"位元"而一群位元則稱爲"字"亦是同樣的情形。只是把某個較難了解的符號用另一個較容易了解的符號來代替而已！

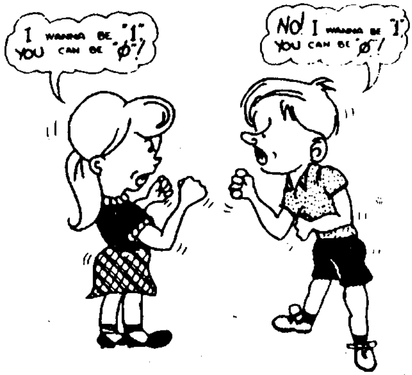
其中翻譯的動作發生在許多個不同的步驟中，它們是

伏特 ⇒ 二元 ⇒ 字元
階層 ⇒ 數位 ⇒ 數目
（+5V，0V） ⇒ （0,1） ⇒ 等等。

注意這些翻譯不是由計算機而是由人來作的！記住，計算機是很笨的！

當我們了解計算機只用伏特值來表示事物後，會有一個問題：我們如何用伏特值來表示事物？結果發現使用二位數字（位元）表示是很簡單的。我們可將兩種伏特值（5V和0V）用兩種二位數字（0和1）代表。你可以設定1等於5V而0等於0V，這設定是任意的。當然也可以令1等於0V而0等於1V。但一般用法都採用第一種的方式。

用一個位元，我們可表示二種不同的值或狀態。如同所謂布林值（眞或假），符號（+或-），是或否，開或關等都是例子。



讓我們來定義一些位元的運算。首先要定義先後次序，因為我們時常要比較二值看看何者較大。“ 0 ”和“ 1 ”當然 1 比 0 大。而其他的二元值常需利用感覺來決定其次序。“ 眞 ” 應該比假大，因此令眞的值爲“ 1 ”（ 或 5V ）而假值爲“ 0 ”（ 或 0V ）。是或否，開或關也可以用同樣方式來設定。

記住 5V 和 0V 的意義隨文中意義而定，5V 有時代表眞，有時代表數字“ 1 ”而有時又代表“ 開 ”等等。從現在開始，我會以“ 1 ”來代表 5V 而 0 來代表 0V 。

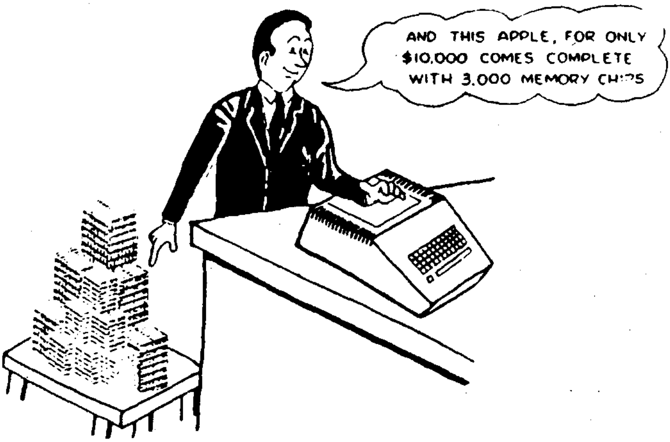
位元字串（ *BIT STRINGS* ）

到目前爲止我們只討論到一個位元的問題。雖然有時候一位元就足夠表示我們所要的資訊了，可是有時候卻需要表示二個以上不同的值。例如十進位中的 0 到 9 ，有 10 個數字，可是一位元只能表示二個值，爲什麼不用比較多的位元來代表呢？例如用 10 個位元來表示 0 到 9 這 10 個數目。“

5 ” 就以“ 0000010000 ” 來表示。“ 0 ” 則令第一個位元爲 1 ，其餘爲 0 來表示。“ 9 ” 則令最後一位元爲 1 ，其餘爲 0 來表示。如下表所示：

DECIMAL DIGIT	BIT NUMBER 0	1	2	3	4	5	6	7	8	9
0	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	0	1

在我們的例子中，一個位元字串中只有一個位元被設定爲 1 ，而 100100100 的值沒有定義，這方法可行但卻效率不夠！雖然每個值都有唯一相對應的位元字串，可是卻有許多唯一的位元字串尚未定義。因爲每個位元都佔用了記憶體



，當然希望在定義位元字串時能最有效地利用記憶體，這樣也就可以降低計算機的成本。

爲容易了解起見，考慮二位元。如前面的設定二個位元可表示二個不同值，等等；前面提過一個位元就可表示二個不同的值，也就是說我們至少浪費了一半的記憶體。爲什麼不定義數目 0 和 1 成二位元如下呢？

值	位元字串
0	00
1	01

再看看下列位元字串

值	位元字串
?	10
?	11

這值尚未定義，當然不能再用 0 與 1 ，因爲 0 與 1 已經等於上列二個位元字串了。

爲什麼不令其等於 2 和 3 呢？如下：

值	位元字串
0	00
1	01
2	10
3	11

如此我們就可用二位元來表示 4 種不同值，比前一種方法節省了二個位元。

假設用三個位元來表示值。如前所示，最左邊的 0 可被忽略不看（左邊的位元通常稱爲高位元），如下：

值	位元字串
0	000
1	001
2	010
3	011
?	100
?	101
?	110
?	111

現在我們尚有 4 個未定值。如前面一般，將它們定義成接下去的 4 個值（ 4 到 7 ）。這樣節省了相當多的記憶體。從 0 到 7 ，這 8 個值只需要使用 3 個位元就可以代表它們了，使用的記憶體約爲前面方法的三分之一！但我們要表示 0 到 9 ，因此勢必要再加上一位元。再加上一位元後就如下表所示：

值	位元字串
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
?	1000
?	1001
?	1010
?	1011
?	1100
?	1101
?	1110
?	1111

再多加一位元，又多了 8 種新值，雖然我們只需要其中的 2 個，但是既然有 16 個值，就可以表示從 0 到 15 。但我們只需表示從 0 到 9 ，所以四個位元組中的 1010 到 1111 就不被定義。這麼做是浪費了一些記憶體，但因爲我們只要表示由 0 到 9 的值，因此這浪費是不可避免的。比起前面 10

個位元，僅用 4 個位元已經節省了不少的記憶體。通常，每多加一個位元，就使組合的數目增加一倍。例如用 8 個位元可表示 256 個值，而 10 個位元可表示 1024 個值，而 16 個位元則可表示 65536 個不同的值！

我們發明了計算機中使用的數字系統。位元字串中的每一位元都代表 2 的乘冪。第一個位元表示 2^0（任何數的 0 次方都為 1），第二個位元代表 $2^1 = 2$，第三個位元為 2^2。例 1100101 代表 $1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 101$。

7	6	6	4	3	2	1	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

用八個位元我們最多可表示（128）+（64）+（32）+（16）+（8）+（4）+（2）+（1）再加 1（因為也可以表示 0）種或 256 種不同的值。通常要表示 $2^n - 1$ 個不同的值需要 n 個位元。如要表示 0 到 9 之間的 10 個數目，3 個位元是不夠的，因為 $2^3 - 1 = 7$，仍然有二個值無法表示，因此要多加一位元，即使這會浪費一部分的位元組合。但是，假如只有 n 個位元則只能表示 2^n 種不同的值（如從 0 到 $2^n - 1$ 之間的數目）。

記住用位元字串亦可以表示不是數目的其他資料。如顏色紅、藍、黃和綠表示如下：

顏色	二進位碼
RED	00
BLUE	01
YELLOW	10
GREEN	11

Or possibly the alphabetic characters:

字符	二進位碼
A	00000
B	00001
C	00010
D	00011
E	00100
F	00101
.	.
.	.
.	.
X	10111
Y	11000
Z	11001
(UNUSED)	11010
(UNUSED)	11011
(UNUSED)	11100
(UNUSED)	11101
(UNUSED)	11110
(UNUSED)	11111

既然有 26 個字母，就需要 5 個位元（ $2^5 = 32$ ），4 個是不夠的（ $2^4 = 16$ ）。

二元算術 （*BINARY ARITHMETIC*）

我們已經知道怎麼樣表示資料了，現在看看如何來處理這些資料。

基本加法規則 （*BASIC ADDITION RULES*）

首先複習一下在十進位時作加法的情形。假如要加 95 和 67 二個數目，加的步驟如下：

首先加 5 和 7

```
  95
+ 67   加5和7
----
   2   結果爲2，進1
```

接著加9和6再加1（進位）

```
  95
+ 67   加9和6和1（由進位而來的）
----
  62   結果爲6，進1
```

最後得到162。

二元加法步驟相同，但更簡單，主要基於七個規則：

1) 0+0 = 0; carry = 0
2) 1+0 = 1; carry = 0
3) 0+1 = 1; carry = 0
4) 1+1 = 0; carry = 1
5) 0+0+carry = 1; carry = 0
6) 1+0+carry = 0; carry = 1
7) 1+1+carry = 1; carry = 1

因此作任何n位元的數目的加法的步驟如下：

第1步）加第一列的0和1，得1，進位爲0

```
0110
0111
----
   1   進位＝0
```

第2步）加第二列的1和1，得0，進位爲1

```
0110
0111
----
  01   進位＝1
```

第 3 步）加 1 和 1 再加 1（進位），得 1，進位為 1

```
  0110
  0111
--------
   101    進位 = 1
```

第 4 步）加 0 和 0 再加 1（進位），得 1，結束運算

```
  0110
  0111
--------
  1101    進位 = 0
```

照此步驟可作任何位元數的加法，如

```
 01101100      1101101
 11101011      1111011
 --------      -------
101010111     11101000
```

不帶符號的整數 （*UNSIGNED INTEGERS*）

到目前為止我們假設想要使用多少位元就有多少位元。但實際上並不是如此，通常我們只有固定數目的位元（ 8 個或 16 個），因此數字的大小有一定的限制。用 16 個位元能表示 0 到 65535 之間的整數（ $2^{6}-1=65535$ ）。用 8 個位元則能表示 0 到 255 之間的值。因為 6502 是 8 位元的微處理機（表示只能用 8 個位元）看起來我們只能處理 0～255 之間的數值，還好，尚有多精確度的處理常規可用。這點以後再討論。不帶符號的整數在定義 0 與 65535 之間，因此一個整數需要 16 個位元。

尼波，位元組和字
(*NIBBLES (NYBBLES?), BYTES, and WORDS*)

在書中我們會常用到長度爲 4，8 和 16 的位元字串。這字串長度並非任意設定而是由所使用的硬體來決定的。6502 習慣令其資料爲 4 ，8 或 16 位元的組合。

“尼波”就是長度爲 4 的一個位元字串。前面提過，要儲存十進位數字至少要使用 4 個位元，因此有時候十進位數字以一個尼波來表示，稱爲二元碼十進位數簡寫成 BCD。在 6502 上可以作二元碼十進位數的運算，以後會討論到。

“位元組”是長度爲 8 的一個位元字串。在 6502 中最普遍的資料型態是位元組，因爲 6502 的資料寬度是 8 位元。（即 6502 是 8 位元的處理機）。

“字”爲長度是 16 的一個位元字串。通常用來存地址或整數。用一字可表示 65536 種不同的值（ 64K ）。因此 6502 就可以直接指到 64K 記憶體的位址了。

一個位元組包含了兩個“尼波”，而一個字包含兩個位元組。每個位元字串都有低層位元和高層位元，低層位元永遠是位元零而高層位元則爲位元 n − 1 。例如，一尼波爲 4 個位元，因此它的高層位元爲位元 3 （記住我們是從位元 0 開始的），而位元組的高層位元則是位元 7 ，字的高層位元爲位元 15 。

範例：

BIT #	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NIB													1	0	1	0
BYTE									1	0	1	1	0	0	1	1
WORD	1	0	1	0	1	1	1	0	1	1	1	0	0	0	1	1

既然在一位元組中包含有二尼波，因此其中一個就稱爲高層尼波而另一個則爲低層尼波。低層尼波由位元 0 到 3，而高層尼波則由位元 4 到 7 所組成。同理一字中的低層位元組由位元 0 到 7 組成而高層位元組則由位元 8 到 15 組成。這個定義待以後我們討論到由八位元組成的資料時會有用。

帶符號的整數（*SIGNED INTEGERS*）

有時候從 0 到（ 2^n-1 ）的範圍不夠表示資料，這時就需多加位元來表示更大的值，可是有時候也需要表示小於 0 的整數。到目前爲止我們討論的數字系統還無法表示負數。因此我們必需重新建立一個新的數字系統。

虽然有许多用于表示负数的系统可用，但因爲 6502 硬體算術單位的限制 2 的補數這樣數字系統。這系統有下列幾種規則：

(1)採用標準二元型式

(2)二元數目的高層位元爲符號，假如爲 1 則表示負數否則爲正數。

(3)假如是正數則與其標準二元型式相同。

(4)假如是負數則以 2 的補數型式來表示。

2 的補數型式是取正數値，再把所有 0 變 1 而 1 變 0，最後再加上 1 即可得！例如數字 2 的 16 位元表示爲

0000000000000010

則 2 的 2 的補數計算方法如下：

首先改變所有位元

1111111111111101

再加上 1

1111111111111110

因此 1111111111111110 就是－2 的 2 的補數表示。我們可把 2 的補數轉換視爲乘以－1。事實上，求負數的 2 的補數結果會得到相對的正數。如－2

1111111111111110

它的 2 的補數爲

0000000000000001

再加 1 得

0000000000000010

即等於 2。

爲什麼要用這麼一個奇怪的方法呢？用高層位元標準型式不更簡單嗎？由下面加法就可以了解這麼作的理由了。2 加－2，其步驟如下：

0000000000000010
1111111111111110
―――――――――――――――
0000000000000000　進位＝1

假如我們不考慮進位，則等於0，這就是我們想要的結果。可以很輕易地證明假如不考慮進位則每次結果都是正確的。

假如不考慮進位，又如何測試是否有溢位（ overflow ）的情形呢？因爲高層位元被視爲符號。因此位元 14 才是眞正有意義的高層位元，從這位元產生進位時就可知道發生溢位情形了！

十六進位數目（*HEXADECIMAL NUMBERS*）

用二元數字讓人們看起來嫌太冗長了！因此在許多年前就有程式設計師把二元數字用八進位數字代表，這麼一來16位元的資訊就變成6個八進位數了。這種八進位數字系統至今仍在一些迷你計算機中通用；而在微電腦產生時又改爲使用16進位數，如此16位元的資料只需4個 16 進位數字就可以表示了。這種方法最大的缺點就是人們尙未能熟悉。16進位數有16種不同數字，0到9是與10進位數相同的而最後6個數字A到F則分別代表10到15的值，就如下面一般：

二進位	十進位	十六進位
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

為什麼要用 16 進位呢？因為它很容易被轉換成二元碼，反之亦然。而十進位數則不然，如 11111100 並不容易被轉換成 252，可轉換成 16 進位的 FC卻是很容易的。要轉換二進位成 16 進位數，首先把 4 個位元分成一組，如果左邊不夠 4 位一組則在前面補 0，然後每 4 個位元就換成相對應的 16 進位數。如前例中 11111100，首先被分成 2 組 4 個位元（ 1111 和 1100 ），其中 1111 等於" F "而 1100 等於" C "，因此 11111100 就等於 FC！

反向轉換也很簡單，把每個 16 進位數換成相對應的二進位數即可。如 EFC4 換成 1110111111010100 。對新的程式員而言或許太麻煩，可是事實上 16 進位數是很方便的。

底數和其毛病
（ *RADIX AND OTHER NASTY DISEASES* ）

現在我們已經認識十進位、二進位和十六進位數。假如看到" 100 "這個數字，如何知道它是以多少為底數呢？是

100 以 2 爲底(4)或是 100 以 10 爲底（ 100 ）或 100 以 16 爲底（ 256 ）呢？

爲避免混淆，通常在前面多加一個符號。假如是二進位則加%， 16 進位則是 $，而 10 進位就是！但 10 進位數前面的區分符號“！”可以省略，所以如果到前面沒有區分符號的數字則是 10 進位數，如此就不會混淆不清了。

ASCII字元集合（*ASCII CHARACTER SET*）

前面提過二元值也可以代表非數值的量。計算機常需要處理文字、數字和某些標點符號組成的文字內容。因此就必須用不同值代表每一字元。

要表示 26 個字母至少要 5 個位元（有 32 個不同的值）。再加上 0 到 9 的數字和小寫字母以及一些標點符號，總共有 96 個字要表示。再加上一些控制機器的字，如游標控制， return ， tab 等共有 128 個字母，因此需要使用 7 個位元。如果允許其他特殊字的加入，則需再多加一位元，那麼總共可以表示 256 種不同的值了。

接著就要指定每個字母所代表的值了。這裡我們不建新的碼而採用美國標準交換資訊碼（ ASCII ）。 ASCII 碼被大多數電腦所採用。 ASCII 碼中前 32 個值被設定爲控制碼，包括 return ， line feed ， backspace ， tab 和其他控制碼。接下來的 32 個值設定爲一些常用的標點符號（如句點、逗點、空白等）和數字，再接下來 32 個值則被設定爲大寫英文字母和較少用的一些標點符號。最後 32 個值設定爲小寫英文字母和罕見的一些標點符號。

ASCII 碼中後面的128個值尚未被定義，它們是留給使用者自行定義的 Apple II 中，剩下的碼由反字和會閃的字組成。見附錄A。

用字元串來表示指令
（*USING BIT STRINGS TO REPRESENT INSTRUCTIONS*）

字元串不僅可用來表示資料，也可以用來表示指令。

假設在日常生活中有一些指令必須遵從。首先是"鬧鐘響"叫你起床，第二個指令是"穿衣服"，第三個指令是"開車去上班"，第四個爲"工作"，第五個指令是"開車回家"，最後則是"上床睡覺"。這些指令可以被設定如下：

字元串	命令
000	Get out of bed.
001	Get ready for work.
010	Drive to work.
011	Perform required duties.
100	Drive home from work.
101	Go to bed.

用這些指令就可以控制一個人的日常生活。但上面的設定並沒有數值的意義，只是表示每次只能作一種動作，而且這些動作可以不以上列順序出現，例如一個人可能"開車上班"，結果發現忘了帶東西而再"開車回家"拿，其動作順序如下：

```
000     Get out of bed.
001     Get ready for work.
010     Drive to work.
100     Drive home and pick up forgotten items
010     Drive back to work.
011     Perform required duties.
100     Drive home from work.
101     Go to bed.
```

使用位元串來表示指令是下面幾章所討論的計算機的基礎。

第三章
暫時器、指令型式和位址

概　論 （*GENERAL*）

到目前爲止，我們討論的資料型態都沒有限制。但實際生活上，卻對資料的大小或運算有某些限制。程式員必須了解 APPLE II 的限制及方便的地方才能寫好程式。

APPLE II 主要有三部分：

(1)中央處理單元（ 6502 微處理機 ）

(2)輸出 / 輸入（ 鍵盤、螢幕、磁碟等 ）單元

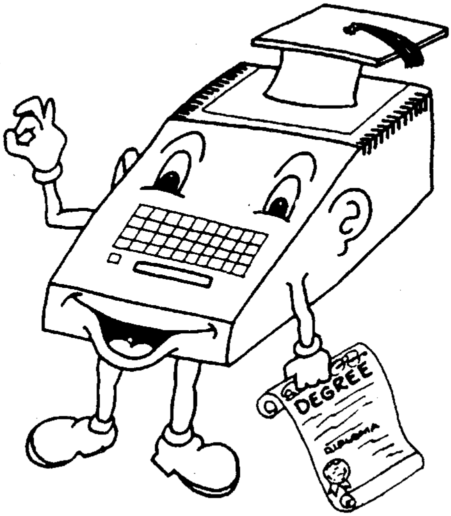
(3)記憶體

APPLE II 上的記憶體爲 65536 個位元組。每個位元組都有位址，也就是說，我們可以隨心所欲地使用 65536個位元組中的任何位元組作運算。

某些（ 事實上爲 5120 個 ）位置在輸出 / 輸入時所使用。其中 1024 個位置由螢幕記憶和其儲存的資料所佔用。（從記憶體位置 $400 到 $7FF ）。另外 4K位置保留給週邊卡片。剩下 59K 位元組用來存放變數程式， BASIC 和 Pascal 等。通常使用者有 48K 的位置可用。

中央處理單元（ CPU ）爲執行指令的地方。 CPU

為電腦的大腦。從記憶體或輸出入機器拿出或送人資料，然後在 CPU 內作運算或比較的工作。



6502 微處理機包括一個算術邏輯單元（ ALU ），所有加法減法等都在這裡執行，和一個控制單元負責從記憶體中拿出或存入資料，解指令碼，同時處理六個特別的暫存器。其中 5 個暫存器佔用 8 個位元（與記憶體相同），另外一個佔用 16 個位元寬（和 6502的地址通道一樣）。每個暫存器都有特別的功能，因此分別有特別的名稱如下：

(1) 累積器（ Accumulator ，A或ACC ）

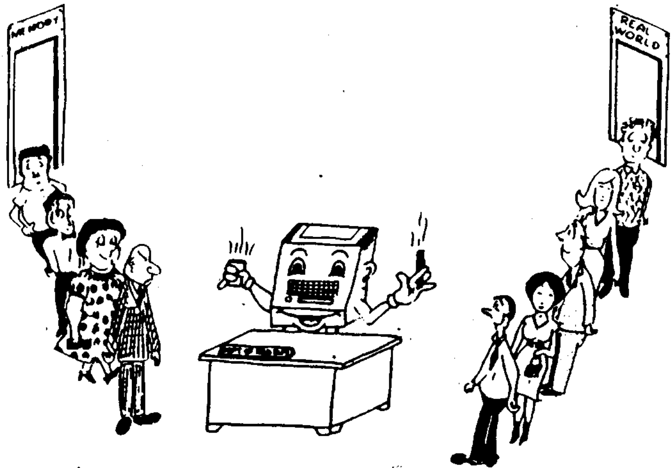
(2) X一暫存器（ X ）

(3) Y 暫存器（ Y ）

(4) 堆疊指標（ SP ）

(5) 程式況狀字（ P或PSW ）

(6) 程式標示（ PC ）



"MEMORY AND REAL WORLD PEOPLE HAVE TO GO THROUGH APPLE TO TALK TO ONE ANOTHER"

累積器（*ACCUMULATOR (A or ACC)*）

大部分資料運算都在累積器上處理。如數字的加減，資料從記憶搬出或搬入也以此爲中間站，所有邏輯運算也在此執行。累積器可說是我們最常使用的暫存器。

X暫存器（*X-REGISTER (X)*）

6502 的X暫存器是有特殊用途的，不能在此作加減它是用來處理陣列（ array ）的元素字串、指標等。利用X暫存器來處理陣列元素稱爲索引。因此X暫存器又稱爲X索引暫存器。

Y暫存器（*Y-REGISTER (Y)*）

与X暫存器一樣也是用來作索引的。有兩個索引暫存器，我們就可以作次字串，串聯和其他陣列的函數的運作了。

堆疊指標(SP)（*STACK POINTER (SP)*）

這個特殊作用的暫存器在呼叫副程式和從副程式跳回主程式時或要保存暫時的資料時將被用到，因爲它只佔用 8 個位元，所以只能指到 256 個位址。（從 $100 到 $1FF。）

注　　意

位置 $100 到 $1FF 保留給堆疊指標來用，所以千萬不要用這些位置來存資料和程式。

程式狀況字　(P或PSW)
（*PROGRAM STATUS WORD (P or PSW)*）

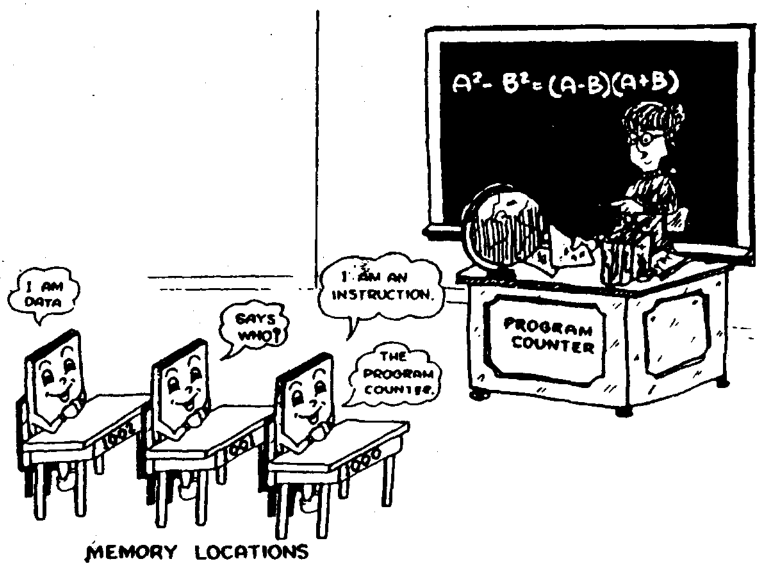
PSW事實上不是暫存器。它是用來儲存 7 種狀況位元，供轉移指令執行時作決定。

程式標示(PC)（*PROGRAM COUNTER (PC)*）

PC 是電腦用來指到現在正在執行的指令的暫存器。是唯一的 16 位元暫存器（因為 16 位元才能指到 APPLE II 上 65536 個不同的位置）。

指令型式(6502)
(*INSTRUCTION FORMAT (6502)*)

指令是用來告訴 6502 應該作什麼動作的。指令是什麼？它只是另外一組存在記憶中的 8 位元碼。因為有 8 位元，所以有 256 種可能的指令在 6502 上，事實上只用到 120 個指令，對應到這 100 到 120 個指令碼稱為有效碼。其餘的稱為無效碼。



運算碼（計算機指令）是同資料一樣存在記憶體中的。

那麼計算機又如何分辨是資料或指令呢？通常記憶中位元組的意義是由上下文來決定的。假如某位元可被程式標示所指到，則此位元組就是計算機指令。我們假設程式是順序存放在記憶體中，也就是說第二個指令是在第一個指令之後，第三個指令之前。

範例：

記憶體

第一個指令　　←程式標示
第二個指令
第三個指令
第四個指令
第五個指令

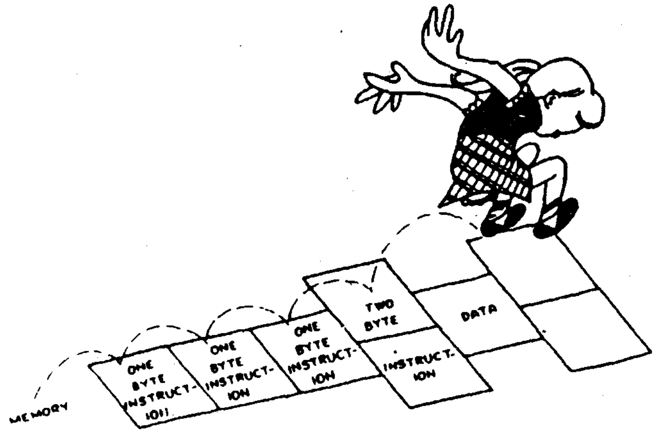
起初令程式標示爲第一個指令的位址處理機就拿此指令且執行，接著程式標示加 1 ，指到下一個指令，處理機再取出來執行，如此重覆下去。

記著計算機無法分辨出是程式或資料。程式標示所指到的就被翻譯成指令。

二位元組和三位元組指令

(*TWO AND 3-BYTE INSTRUCTIONS.*)

有許多指令需要作用不只 1 個位元。例如要把一個 8 位元的常數放入累積器中，要如何來指明這常數呢？直接加到指令後面就可以了；當 16 進位碼 $A9 執行時， 6502 就把下一位元組的常數放入累積器中。一個指令加一個常數所以是二位元組，當然作完之後，程式標示要加 2 而不是加 1 ，否則常數會被當作指令來執行。



"PROGRAM COUNTER GETS TO SKIP DATA AFTER A TWO BYTE INSTRUCTION"

除了二位元組指令外，還有三位元組的指令。例如把"累積器中的值存到記憶體中某位置上"，這個指令由 $8D和一個 16 位元的位址組成。如（ $8D, $00, $10 ）會把累積器中的值存在位置 $1000 上而（ $8D, $C3, $48 ）會把值存在 $48C3 位置上。

記住當碰到多位元組指令時，程式標示會自動增加跳過多加的資料。

範例：

```
A9   INSTRUCTION #1   LOAD ACC WITH $FF
FF

8D   INSTRUCTION #2   STORE ACC AT LOCATION $1234
34
12

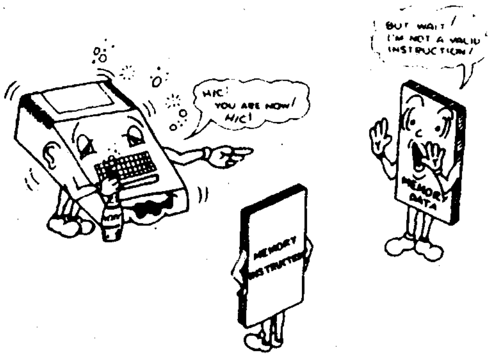
--   ETC.

--

--
```

警　告

計算機無法分辨資料和有效指令。在上面的例中，假如程式從位置$1234開始，則首先把$FF 放入累積器中，接著會破壞第一個指令（因爲第一個指令放在位置$1234上)。



下面的碼即爲實際情形：

LOC	DATA/CODE
1234	FF
1235	FF
1236	8D
1237	34
1238	12
1239	—
ETC.	ETC.

要小心地找位置存資料，否則可能會破壞你的程式。

6502的位址型式（*6502 ADDRESSING MODES*）

6502 微處理機有 56 種指令，可是前面提過約有 120 種不同的指令碼，爲什麼不同呢？因爲有些運算有好幾種運作法。例如 6502 中有一種運算爲"把一個 8 位元值放入累積器中"，這運算 ou 作"載入累積器運算"，簡寫成 LDA 。 LDA 有好多種可以把常數或記憶體中某位置的值或陣列的某一元素或字串放入累積器中。這幾種運算有共同點：將 6502 累積器中存入一個新值。雖然動作相同（存入累積器中），可是方法不同，因此 6502 就用不同的運算碼來代表每種 LDA 運算。這種不同稱爲"位址型式"。指令告訴計算機要作什麼而位址型式則指明到那裡去拿資料。

直接位址型式
（*IMMEDIATE ADDRESSING MODE*）

這型式告訴計算機所用的資料爲 8 位元常數，緊接在指令之後。在前面例子中" $A9 "表示"把下一個位元組的值放入累積器中"。這就是直接位址型式。這種型式的指令通常都是佔用二位元組：一爲指令，一爲直接的資料。

絕對位址型式
（*ABSOLUTE ADDRESSING MODE*）

有時候要把記憶中某位置的值放入累積器中，需要一位元組來說明指令（ LDA ），其次要指明是 6502 中 65536 個位置中的那一個，需要二位元組位址。這種型式稱爲"絕對位址型式"，指令就需要三位元組來表示了：一爲指令，兩個位元組作爲位址。這種型式的 LDA 實際上的指令碼爲

$AD，接著為兩個位元組表示位址；先為低階位址，後為高階位址。例如，要把位置 $1234 的值存入累積器中，則指令碼應為（ $AD, $34, $12 或 AD3412 ）。

零頁位址型式
（*ZERO PAGE ADDRESSING MODE.*）

6502 中有一種與絕對位址型式很類似的"零頁位址型式"。亦是把某一記憶體位置的值放入累積器中，唯一不同點是這種指令只佔用二位元組：一為指令，一為位址。一個位元組只有 8 個位元，因此只准有 256 個不同位址，在 6502 中它們對應第零頁上的 256 個位址。為什麼要用這種限制較多的型式呢？原因有二，一為它只需二位元組，所以可省記憶位置，第二個原因是這種型式較絕對位址型式的速度快。第零頁通常被用來放變數，而其他位置則用來放程式，陣列和字串。

索引位址型式
（*INDEXED ADDRESSING MODE.*）

前面說過 X 暫存器和 Y 暫存器是用來作索引暫存器。索引暫存器是用來處理陣列的元素或字串。記得在整數 BASIC 中是在陣列名稱後加個括號圍成的索引來指明是那個元素（如 M(I)：I 為索引）。X，Y 暫存器就用來代替 I。例如：指令碼 $BD 告訴 6502，把下二個位元組再加上 X 暫存器中的值當作位址，把位址內的值放入累積器中。例如指令碼 BD 34 12，而 X 暫存器中的值為 5。就表示把 1234 + 5 =

1239位置內的值放入累積器中。

注　　意

Y 暫存器的用法一樣，只是指令碼不同，但效果相同。

間接位址（*INDIRECT ADDRESSING.*）

這種型式較複雜，把指令的下二位元組所指的位置中的值作爲低階位址，而高階位址則到指令的後 16 位元位址再加 1 所指到的位置上去拿。待會兒看些例子，會更容易了解。

由 Y 間接索引（*INDIRECT INDEXED BY Y.*）

與間接索引相似，詳細情形待會兒再討論。

間接的由 X 索引（*INDEXED BY X. INDIRECT.*）

下面再討論。

暗示的位址型式
（*IMPLIED ADDRESSING MODE.*）

即指令本身會表示所用的資料是那種型態。這種指令都

是一位元組長

累積器位址型式
（*ACCUMULATOR ADDRESSING MODE.*）

即運算是對累積器中內容所作的這種指令也是一位元組長。

注　意

可以看出 6502中許多運算都是屬於累積器位址型式。眞正的累積器位址型式與其他型式不同之處在其只用累積器而不需從記憶體中取運算元。

相對位址型式
（*RELATIVE ADDRESSING MODE.*）

這種型式用在所謂跳開指令中。下面再詳細討論。

結　語（*ADDRESSING MODE WRAP-UP.*）

假如你尚未了解這些位址型式，沒關係，這只是簡介，下面會詳細討論各種位址型式的。

第四章

一些簡單指令

新指令：

```
EQU EPZ DFS
LDA LDX LDY STA STX STY INC DEC
TAX TAY TXA TYA INX INY DEX DEY
```

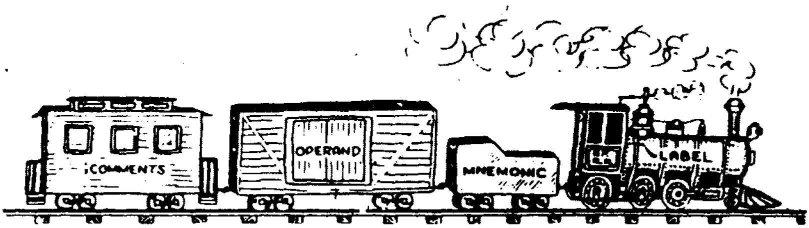
概　論（GENERAL）

到目前爲止，我們都用碼來表示指令，但總共有120種不同的指令碼，要記住多麼困難。用片語來代替指令碼就好多了。LISA 就是接受一些片語然後把它們翻成指令碼的程式，這種程式稱爲組合編譯器。我們不用長的片語，代之以短的符號來代表。符號都是三個字母，例如 LDA 代表"載入累積器中"，STA 代表"儲存累積器"。雖然必須學習這些符號，但卻值得。當輸入程式時，只需打入三個字，而不需打入整個句子了！

組合語言的程式型式
（*ASSEMBLY LANGUAGE SOURCE FORMAT*）

6502 所了解的機器語言碼通常稱為“結果碼”（object code.）。而人們所了解的符號則稱為文字檔案或原始碼（source code.）。

跟 BASIC 的不限制陳述的安排不同，組合語言有固定型式。每一個組合語言陳述被分成 4 欄，分別是標籤欄，符號欄，運算元欄和註解欄。各欄以空白來隔開，其中有二、三欄為可有可無。



LABEL MNEMONIC OPERAND ;COMMENTS

標籤欄的值為原始程式中某一行的標籤，相當於 BASIC 中的行數。所有跳開的指令（BASIC 中 GOTO）會用到標籤。但此處的標籤是個字串，不像 BASIC 中為數字，字串的長度為 8 且開頭第一個字一定是個大寫字母。標籤可由大寫字母和數字所組成。

有效的標籤例子：（EXAMPLES OF VALID LABELS）

```
LABEL
LOO1
A
MONKEY
A.B.C
```

無效的標籤例子：

```
1HOLD       (BEGINS WITH "1")
HELLOTHERE  (LONGER THAN 6 CHARS)
LBL.X       (CONTAINS ".")
```

標籤不像BASIC中的行數要每行都有，只有當你要用到某一陳述時才需要標籤，且必須從第一格開始寫，寫完之後要空一格再寫符號。

標籤可有可無，假如某一行沒有標籤則第一格一定要是空白。

符號欄　（*MNEMONIC FIELD.*）

標籤欄之後爲符號欄，其中爲LISA所規定的三個字母的符號，如LDA, LDX, LDY, STA……等。

運算元欄　（*OPERAND FIELD.*）

此欄在符號欄之後，包括位址和位址型式。假如沒有位址型式，則表示用絕對位址型式。位址可以是個"位址式子"，和BASIC中的算術式子類似，只是這裡只能用加減法。例如，$1000 + $1 等於 $1001，假如指令爲"LDA $1000 + $1"，則表示記憶體位置$1001上的值載入累積器中。待會再詳細討論位址式子。

假如要指定常數，則在 16 位元的位址前面加個"#"

或“ / ”，若是“ # ”則位址式子的低階 8 位元作爲常數，若是“ / ”則高階 8 位元作爲資料値使用。

若是索引位址型式則在位址式子後面加個“ X ”或“ Y ”，決定是以 X 爲索引或以 Y 爲索引。

若是暗示性位址型式或累積器位址型式，則運算元欄應爲空白，否則就會發生錯誤。

至於間接，由 Y 間索引和間接由 X 索引的句法以後會討論。

註解欄 （ *COMMENT FIELD.*）

在運算元欄之後可以加上一些對此行指令的解釋，但注意一定要以“ ; ”爲開頭且與前欄之間至少有一格空白。

載入羣 （ *LOAD GROUP.*）

這類指令有三種： LDA （ 載入累積器 ）， LDX （ 載入 X 暫存器 ）和 LDY （ 載入 Y 暫存器 ）。這些指令到運算元欄所指明的位置上取出其值，然後放入所指定的暫存器中。例如 LDA $ 1FA0 從位置 $1FA0 中取值載入累積器中。注意所指定位置上的値並沒有改變，只是抄了一份放入累積器中而已。通常 LDA $nnnn （ 其中 nnnn 爲 1 到 4 位數字的 16 進位數 ）會把位置 $nnnn 的値載入累積器中。

範例：

LDA $11F0 ——把位置$11F0的值載入累積器中

LDA $127F ——把位置$127F的值載入累積器中

LDA $0 ——把位置$0 的值載入累積器中

把常數載入累積器中，只要在常數之前加個符號"#"或"/"來說明要載入運算元欄中的16位元式子中的低階的8位元或高階的8位元即可。

範例：

LDA #$1000 ——把值$00載入累積器中（$00爲$1000 的低階位元組，而高階位元組$10被忽略不管）。

LDA #$FF ——把值$FF載入累積器（$FF原來是爲$00FF，但高階位元組被忽略）。

LDA /$1000 ——把值$10載入累積器中，$10爲值$1000的高階位元，因爲有"/"符號，所以取高階位元而忽略低階位元。

LDA /$FF ——把值$00載入累積器中，$00爲值$00FF的高階位元組。

LDA /$0 ——把值$00載入累積器中。值$0的高階位元組，和低階位元組都是$00。

上面例子都是把值載入累積器中，也可以把值載入X暫存器或Y暫存器中，指令爲LDX和LDY。除了絕對和直接

位址表示方法外，還有其他位址法可用在載入暫存器指令中。

存入指令（*STORE INSTRUCTIONS.*）

我們也可以把累積器和暫存器的值存入記憶體中。這種指令有三種：STA（從累積器存入），STX（從X暫存器存入）和 STY（從Y暫存器存入）。在指令後面要加上所要存放記憶體的位址。

範例：

STA $1000 ——把累積器的值存在位置$1000上，累積器的值不變。

STA $2563 ——把累積器的值存入位置$2563上，累積器的值不變。

STA $FF ——把累積器的值存入位置$FF上。

同樣的，可以使用 STX 和 STY 指令從X，Y暫存器把值存入記憶體中。

STX $1500 ——把X暫存器的值存入位置$1500上，X暫存器的值不變。

STY $220 ——把Y暫存器的值存入位置$220上。

注意，存入指令並不會改變累積器和暫存器的值。

現在我們用學過的一些基本指令來寫一個簡單的組合語言程式。這程式只是把位置 $1000和$1001 上的內容移到位

置 $2000 和 $2001 上。程式執行完之後，位置 $2000 和 $ $1000 有相同的值，位置 $2001 和 $1001 也有相同的值。最簡單的方法是直接讓位置 $2000 等於位置 $1000，而位置 $2001 等於位置 $1001，可是並沒有這種指令可用，只好先把位置 $1000 的值移到累積器中，再從累積器移到位置 $ $2000 上，同理也把位置 $1001 的值移到累積器，再從累積器移到位置 $2001 上。程式如下：

```
LDA $1000
STA $2000
LDA $1001
STA $2001
```

所有資料轉移都要經由 6502 的暫存器，通常都用累積器。

資料轉移指令
(*DATA TRANSFER INSTRUCTIONS*)

6502 除了提供暫存器和記憶體之間交換資料的指令外，也有暫存器彼此交換資料的指令，如從累積器移到 X，或 Y 暫存器，反之從 X，Y 暫存器移到累積器中。此外還有從 X 暫存器移到堆疊指標和從堆疊指標移到 X 暫存器的指令。符號如下：

TXA ——從 X 暫存器將值移到累積器
TYA ——從 Y 暫存器將值移到累積器
TAX ——從累積器將值移到 X 暫存器
TAY ——從累積器將值移到 Y 暫存器
TXS ——從 X 暫存器將值移到 SP（堆疊指標）
TSX ——從 SP 將值移到 X 暫存器

或許你已注意到沒有直接從X暫存器移到Y暫存器或反方向的指令。可用下列指令來完成這個動作：

從X到Y	從Y到X
TXA	TYA
TAY	TAX
－或－	－或－
STX $nnnn	STY $nnnn
LDY $nnnn	LDX $nnnn

暫存器轉移指令不需要運算元，這就是暗示性位址法的一種，指令本身已指定要在那個位置上作運算。

暫存器的增加和減少
（*REGISTER INCREMENTS AND DECREMENTS*）

接著要討論一些在暫存器上作運算的指令。首先爲X，Y暫存器的增加和減少等4個指令。（增加表示加1，減少表示減1）

INX ——把X暫存器的值加1，值仍留在X暫存器中
INT ——把Y暫存器的值加1，值仍留在Y暫存器中
DEX ——把X暫存器的值減1，值仍留在X暫存器中
DEY ——把Y暫存器的值減1，值仍留在Y暫存器中

由於常常要把暫存器的值加1或減1，因此這些指令很方便。

有個問題即當暫存器的值爲$FF，而又要加1，或值爲

$00 而又要減 1 時會怎麼樣的情形呢？當值爲 $FF 又要加 1 時，計算機會讓其進位但只取後面的值 $00，同理若值爲 $00，又要減 1 時，會讓其退位，也取後面的值 $FF。

INX, INY, DEX 和 DEY 也是暗示性位址法，因此不需要運算元。

增加和減少指令（*INCREMENT AND DECREMENT INSTRUCTIONS*）

INC 和 DEC 二指令不需經由暫存器或累積器，而可以直接在記憶位置作運算。

範例：

INC $2255 ——把位置 $2255 的值加 1，結果仍存在位置 $2255 上。

DEC $15 ——把位置 $15 的值減 1，結果仍存在位置 $15 上。

INC 和 DEC 並不是暗示性位址法，因此需要有運算元，它們與載入和存入指令一樣爲絕對或第 0 頁位址法。記住只有 8 位元，假如要增加 $FF 或減少 $00 則會只取後面的值。

標示和變數（*LABELS AND VARIABLES*）

到目前爲止，要用變數時，都要指明確實的位址，這不太方便。例如用 XCOORD 來代表 X 軸上某一點的坐標要比

$800 來代表更有意義。標示允許我們可用 LDA XCOORD 來代替 LDA $800！在程式某位置上，定義某標示等於某值，以後用這標示就代表這個值。在上例中可定義XCOORD等於 $800 ，則 LDA XCOORD 就相當於LDA $800 ，組合編譯器會自動將XCOORD替換成 $800 。因此就可用標示來代替位址。你可以用標示給記憶位置較有意義的名字。

但如何令標示等於某值呢？很簡單，可用 EQU 假指令。什麼是假指令（ pseudo instruction ）？是在程式中用來給 LISA 的指令。叫 LISA 根據後面的資料作某些事情！假指令不會產生供 6502 微處理機用的碼！

EQU 假指令的寫法如下：

```
LABEL  EQU  <value>
```

標示和值都要。 EQU 假指令告訴 LISA 把標示和相對的值存入符號表中，以後在程式中用到這標示時， LISA 會在符號表中找出這標示，且用其值來替換這標示。如此你就不用記住難記的位址，只要記住所用的變數名稱（或標示）就可以了！

範例：

```
XCOORD  EQU  $800    ——令 XCOORD 等於 $800
LABEL   EQU  $1000   ——令 LABEL 等於 $1000

        LDA  XCOORD——和 LDA  $800 一樣
        STA  LEBEL  ——和 STA  $1000 一樣

CONST   EQU  $FF22   ——令 CONST 等於 $FF22
```

LDA #CONST——把值 $22 載入累積器中（$22 爲 CONST位置上的低階位元組）

LDA /CONST——把值 $FF 載入累積器中（$FF 爲 CONST位置上的高階位元組）

INX XCOORD——把位置 $800 上的值加 1

DEC LABEL——把位置 $1000 上的值減 1

用 EQU 來定義標示時，一定要用絕對位址法。如果要用第 0 頁位址法則用另一指令 EPZ。EPZ 的文法和 EQU 相同。

範例：

```
LABEL EPZ <value>
```

值必須小於等於 $FF。

但定義變數時要擔心資料到底要存在那裡，仍不方便。假如可以能只告訴 LISA 說我們需要一位元組的變數，讓它幫忙找實際的位置就更好了。假指令 DFS 就可以作到，其文法如下：

```
LABEL DFS <value>
```

這裡不說資料要放在那裡而說明要保留多少位元組給你的變數。通常是一或二個位元組。

DFS 令現在的碼位置作爲變數的位址，因此在程式中

要小心地使用 DFS 假指令，以免被當成指令來執行。

運算元欄中的例子
(*EXPRESSIONS IN THE OPERAND FIELD*).

假設上面例子中XCOORD爲在位置 $800 和 $801 上的16位元值 LISA 允許運算元欄中可有簡單的算術運算如“ + ”和“ − ”。

範例：

```
XCOORD  EQU  $800
        LDA  #$0          —清除累積器
        STA  XCOORD       —清除位置 $800
        STA  XCOORD + 1   —清除位置 $801
```

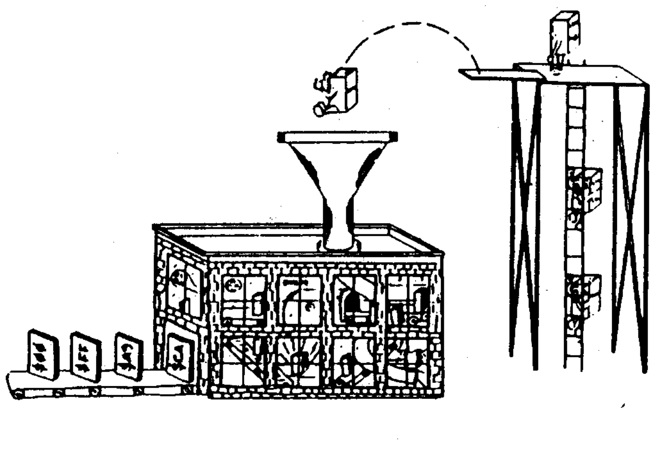
有些 LISA 的版本還允許用乘法，除法和一些邏輯運算。

第五章

組合語言

新指令：

```
BRK JMP BCC BCS BEQ BNE BMI CLD
BPL BVC BVS BLT BGE BFL BTR SED
CMP CPX CPY CLC CLV SEC CLI SEI
END
```



組合編譯器

概　論（GENERAL）

載入和存入指令都是順序執行的指令，在作完此指令之後就接著作下個指令。但有時候需要改變程式的流程到其他地方去執行。組合語言中並沒有 GOTO, FOR/NEXT, 或 IF /THEN 這些指令可用，6502 中只有一些跳開或轉移（jump and branch）指令可用。

最後必須能叫計算機停止最簡單的方法是 BRK 指令，當碰到 BRK 指令時會響起鈴聲，然後把控制權交給 APPLE 監督程式。BRK 指令有個好處，在回到監督程式之前會把 6502 暫存器的值印出，這是一種很簡單的輸出型式。

示範程式（EXAMPLE PROGRAM）

現在用載入和 BRK 指令來寫一個程式。首先叫出LISA，接著作下列步驟：

(1)當出現"！"符號時，打入 INS，然後按 CR 鍵。

(2) LISA 在下行中會送回一個"1"。

(3)打入一個空白，LDA #$0，然後按 CR 鍵。

(4) LISA 會在下行中送回一個"2"。

(5)打入一個空白，LDX #$1，然後按 CR 鍵。

(6) LISA 會在下行中送回一個"3"。

(7)打入一個空白，LDY #$2，然後按 CR 鍵。

(8) LISA 會在下行中送回一個"4"。

(9)打入一個空白，BRK，然後按 CR 鍵。

(10) LISA 會在下行中送回一個"5"。

(11)打入一個空白， END ，然後按 CR 鍵。

第 11 步驟告訴 LISA 已經到了。程式的結束部分。

(12) LIS 會在下行中送回一個“ 6 ”。

既然已經寫完程式。

(13)打 ` control-E ，然後按 CR 鍵。

(14)在螢幕的下一行會出現“ ！ ”。

整個螢幕如下：

```
!INS
   1    LDA  #$0
   2    LDX  #$1
   3    LDY  #$2
   4    BRK
   5    END
   6
```

LISA 現在等著你打入下一個命令（ command ）。在執行程式之前要先把程式翻譯過，在“ ！ ”之後打入“ ASM ”即可。在翻譯過程中， USA 會把程式顯示在螢幕上。當“ ！ ”再出現時打入“ BRK ”，回到 Apple 監督程式中。在“ * ”出現時，打入“ 800 G ”再按 CR 鍵就可以執行程式了！此時麥克風會發出聲音而螢幕出現一些訊號如下：

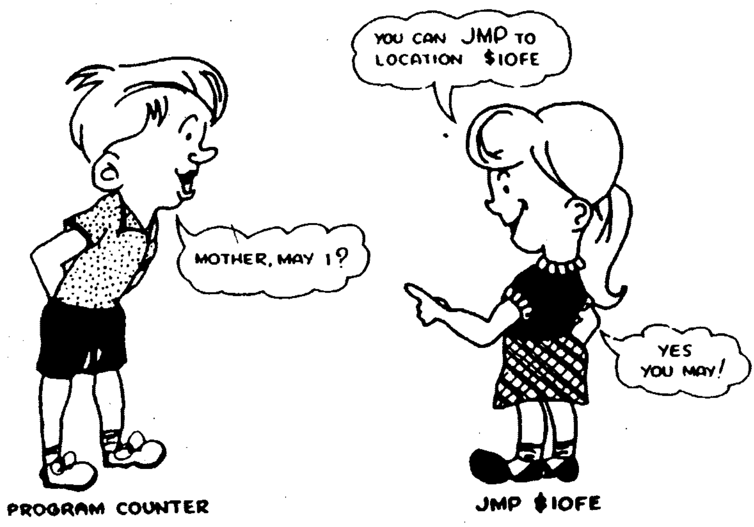
```
0808-    A=00 X=01 Y=02 P=30 S=F0
```

0808 是 BRK 指令在記憶體中的位置加 2 。也就是說是 BRK 指令被放在位置 $0806 上。在你作程式偵錯時就會明白爲什麼要加 2 。

接著 5 個值是當執行 BRK 指令時，累積器， X ， Y 暫存器和堆疊指標的值。在本質上， BRK 指令和 BASIC 中的 END 和 STOP 陳述很像。

跳開指令(JMP INSTRUCTION)

6502 的 JMP 指令是無條件的轉移(branch)，和 BASIC 中的 GOTO 作用相同，只是在 BASIC 中只指明跳到那一行而在此則是跳到一個絕對位址。下列不斷的迴路(infinite loop)一直把位置" J "的值抄到位置" I "上，再把位置 J 的值設為 0 。



範例：

PROGRAM LOC.		STATEMENT
$800	I	EQU $0
$800	J	EQU $1
$800		LDA J
$803		STA I
$806		LDA #$0
$808		STA J
$80B		JMP $800
$80B		END

（注意 EQU 和 END 指令不會佔記憶位置）

JMP 指令都是 3 位元組長：JMP 指令碼，接著是目的地的低階和高階位址。

用絕對位址有個問題。首先，在寫程式時，往往還不知道 JMP 指令的目的地的位址，可用標示來作為 JMP 指令的目的地位址，就和在載入，存入指令中用標示代替位址時一樣。但用標示時，最後是不是仍要用 EQU，EPZ，或 DFS 來定義標示呢？不必！假如標示和指令在同一行，且由第一格開始，則這指令的位址就作為此標示的值。前面的程式可改成如下：

```
I       EQU $0
J       EQU $1
LABEL   LDA J
        STA I
        LDA #$0
        STA J
        JMP LABEL
        END
```

你可以發現這比在 BASIC 中用 GOTO 更方便，因為不必麻煩用順序行數，尤其是往前跳時。編譯器察看一行的第一格是否有標示。假如在第一格中有一大寫字母，則其後的字母（直到空白或“；”之前）都視為標示的一部分。在標示和符號之間至少要有一格空白。假如這一行沒有標示，則第一格一定要是空白，否則編譯器會把符號當作標示，而把運算元（假如有）當作符號。結果會發生錯誤！因此記住，如果沒有標示，則一定要在第一格留個空白。

處理機狀況暫存器
(*PROCESSOR STATUS REGISTER (P or PSW)*)

6502 並沒有 IF/THEN 或 FOR/NEXT 指令。要測試情況則可測試處理機狀況暫存器（ PSW ）中的位元。

PSW和 6502 其他暫存器不同，其他暫存器都是 8 位元或 16 位元暫存器，其中的資料都被看成 8 位元或 16 位元的資訊，但 PSW 則是 8 個單獨位元的集合。（事實上只用 7 個位元，其中一位元不使用）。

其中四位元由前一個指令的結果來設定值。例如 P 暫存器中有一個零旗標，假如上個指令的結果爲 0 ，則此旗標被設爲 1 ，否則爲 0 。其餘二個旗標由 6502 指令來設定，最後一個則表示最近的中斷是由 BRK 指令產生的。

BREAK旗標(*BREAK FLAG (B)*)

打斷旗標（ PSW 中的位元 4 ）只有當上次中斷是由 BRK 指令所產生時才設定。你可以發現當執行 BRK 指令時，P 暫存器的值常常是 30 （有時候會是其他值）把 30 轉換成 16 進位數則可發現位元 4 永遠是 1 ，因爲上個執行的指令是 BRK 指令。

十進位旗標(D)

十進位旗標是由 SED (set decimal) 指令來設定的，而由 CLD (clear decimal flag) 指令來去除。這個旗標用來決定 6502 微處理機要使用那一種基底作算術。下一章會再詳細討論。

廢除中斷旗標 (*INTERRUPT DISABLE FLAG (I)*)

中斷不在本書討論範圍中。但在 PSW 中有一旗標是用來防止中斷的發生（PSW中的位元 2）。這個旗標由指令 SEI 來設定而由 CLI 指令來去除。當這旗標被設定時，6502 的 IRQ 線就失去效能。

情況碼旗標 (*CONDITION CODE FLAGS (N,V,Z,C)*)

情況碼旗標是由正常 6502 的運算結果未設定。當上次運算結果爲零時，Z 旗標就被設定；例如把 $0 載入累積器中，把值爲 1 的暫存器減 1 等指令都會設定 Z 旗標。並沒有特別用來設定或去除 Z 旗標的指令；假如想要設定 Z 旗標，只要把 0 載入暫存器，而要去除則把不是 0 的值載入暫存器即可。假如不想改變暫存器的值，則只要把某值爲 $FF位置再加 1 即可，在 Apple 監督程式中，位置 $FF 的值就是 $FF。使用指令" INC $FFC3 "就會使 Z 旗標被設定。同理要除去 Z 旗標，則把值不是 $FF的位置加 1，如位置 $F800，即可。Z 旗標在 PSW的位元 1。

假如前面一個運算結果爲負數，則 N 旗標就被設定。但

6502 暫存器中並沒有負值。而是以 2 的補數法來表示負數，且以高階位元作爲正負符號，如果高階位元爲 1 則表示是負數，否則爲正數。因此 N 旗標就等於上次結果的位元 7 的值。N 旗標在 PSW 中的第 7 個位元。

我們也沒有特定的指令來設定或除去 N 旗標。如果要設定 N 旗標，可以把值在 $7F 到 $FE 之間的位置加 1 即可。這運算的結果都是負數，也就是位元 7 會是 1 。APPLE 監督程式中位置 $F804 的值就在此範圍內。同理要除去 N 旗標只要把值在 $00 和 $7E 之間，或值爲 $FF 的位置加 1 ，結果會是正數，也就是位元 7 爲 0 即可。監督程式中位置 $F800 的值即是 $FF，這是一個好的選擇。

進位旗標(C)受加法，減法，比較和邏輯運算所影響。同時也可特別由指令 SEC 和 CLC 來設定或除去旗標。C 位在 PSW 的位元 0 。

最後一個旗標是溢位（ overflow ）旗標(V)。這旗標用在有符號的運算中，且受加法，減法和位元測試運算所影響。同時也可由指令 CLV 來除去，但沒有特別的指令可用來設定溢位旗標。在 PSW 中是位元 6 。

在 PSW 中沒有用到的位元是位元 5 ，通常值爲 1 ，但不能保證。沒有 6502 指令用到此旗標。

可試著執行下列程式，注意對 P 暫存器的影響。

```
PGM1:
        LDA #$0
        BRK
        END
PGM2:
        LDA #$1
        BRK
        END
PGM3:
        CLC
        BRK
        END
```

```
PGM4:
        SEC
        BRK
        END
PGM5:
        LDA  #$80
        BRK
        END
PGM6:
        LDA  #$7F
        BRK
        END
```

轉移指令（*BRANCH INSTRUCTIONS (6502)*）

了解了某些運算對 PSW 中的旗標的影響之後，就可用來模擬 BASIC 中的 IF/THEN 陳述。因爲 6502 允許某些分歧指令可以測試這些旗標。

轉移指令和跳開指令很像都是改變程式的流程，可是跳開指令是無條件地跳開，而轉移指令卻是有條件的，在跳開之前，先作測試，測試 PSW 中某一旗標，假如符合條件則跳開，否則執行下一個指令。

轉移指令可用來測試任一種情況碼旗標，看它們是被設定或除去。下列爲允許的轉移指令：

BCC ——假如進位旗標被除去則跳開。
BCS ——假如進位旗標被設定則跳開。
BEQ ——假如 Z 旗標被設定則跳開。
BNE ——假如 Z 旗標被除去則跳開。
BMI ——假如 N 旗標被設定則跳開。
BPL ——假如 N 旗標被除去則跳開。
BVS ——假如 V 旗標被設定則跳開。
BVC ——假如 V 旗標被除去則跳開。

這些指令和 JMP 指令一樣需指定一位址（或標示）。

範例：

```
            LDA #$0
            BEQ LBL1
LBL2        LDA #$FF
LBL1        BEQ LBL2
```

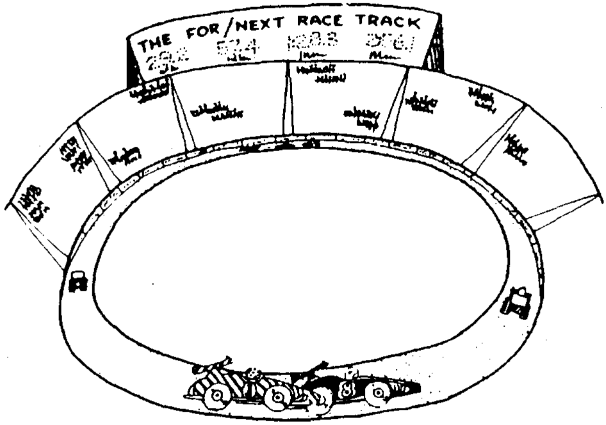
上例中，0 載入累積器中，這使得 Z 旗標被設定而使下面的轉移指令跳開到 LBL1 ，而在 LBL1 又有另一轉移，假如為 0 則跳開的轉移指令。因為並沒有改變暫存器或記憶體內的值，因此 Z 旗標不會改變，也就是分跳到 LBL2 上執行把值 $FF 載入累積器的指令，下個指令（即在 LBL1 上的指令）為若 Z 旗標被設定則跳開，因為剛剛把值 $FF載入累積器中，Z 旗標被除去，所以不會跳開而接著作 LBL1 下面的指令。

假如你想測試看看某個位置上的值是否為 0 ，然後再加 1 ，那麼可用下列指令：

```
    LDA $1F     ;GET THE VALUE CONTAINED IN LOCATION $1F
    BNE LBL     ;IF IT IS NOT ZERO BRANCH TO "LBL"
    INC $1F     ;ADD ONE TO THE VALUE AT LOCATION $1F
LBL ---         ;NEXT INSTRUCTION
    ETC.
```

迴路（*LOOPS*）

計算機最有用的特性之一就是可以反覆不斷地執行某些指令到你滿意為止。這種技巧稱為迴路。在 BASIC 中可用 FOR/NEXT指令，但在組合語言中沒有這種指令。



"PROGRAM LOOPS"

最簡單的模擬法是把次數值載入累積器中，然後減1，直到變成0爲止。用BNE指令可使迴路體（ body ）的指令被執行直到累積器變成0爲止。

例如要把變數J加10 。到目前爲止尙未提到加法，但可用增加來作。因爲INC每次只加1，因此必須加10次。當然可以寫10個INC指令，但是不太方便。我們可以把10存入某位置（例如I），然後建立一迴路來加 10 個1到J變數。程式如下：

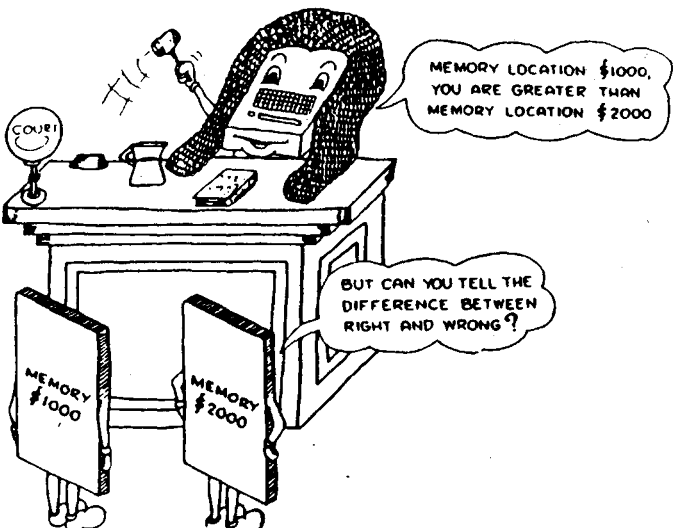
```
I    EQU 0
J    EQU 1
     LDA #!10      ;INITIALIZE I TO 10
     STA I
     LDA #$0       ;INITIALIZE J TO 0
     STA J
LP   INC J         ;NOW, INCREMENT J 10 TIMES
     DEC I
     BNE LP
     LDA J         ;LOAD J SO WE CAN DISPLAY IT
     BRK           ;BREAK AND DISPLAY J (IN THE ACC)
     END
```

比較 （COMPARISONS）

在實際應用時，除了情況碼旗標之外還常需要測試其他事物。例如有時要測試 I 是否等於 5 或（X = 6）AND（j <=（I × 5 + 2））OR（L = M）等情況。因此 6502 提供了比較指令。

CMP 指令比較累積器和運算元位置上的值。如何比較呢？把累積器的值減去運算元的值，根據結果來設定 PSW 旗標。在指令作完之後累積器和記憶體位置的值都不變。假如累積器的值等於指定位置上的值，則相減的結果爲 0，可用 BEQ 指令來測試相等。同理，如果二者值不等，則 Z 旗標必爲 0，可用 BNE 來測試此情況。

N 和 C 旗標亦同樣地受影響。假如 C 旗標爲 1 則表累積器的值大於，等於指定位置的值，否則累積器的值就小於指定位置的值。



V 旗標不受比較指令的影響，因此在比較指令之後用

BVC 或 BVS 指令並沒有用意。

其外也可以用 CPX 和 CPY 指令來比較 X，Y 暫存器和某記憶體的值。情況碼旗標的改變與 CMP 指令相同，也可用轉移指令來測試各種情況。

6502 微處理機中沒有 BGT（ (branch if greater than) an ）和 BLE (branch if less than or equal) 指令。但是可用 BEQ，BNE, BLT, , BGE 指令來組成同樣功效的指令？假如要比較 I 和 J 的值，若 I ≤ J 則跳到 LBL，程式如下：

```
LDA I
CMP J
BLT LBL
BEQ LBL
```

假如 I < J，則第一個轉移指令會跳開至 LBL，假如 I = J 則不會跳開，但第二個轉移指令（ BEQ ）會跳開至 LBL 。

測試大於的函數稍困難些。比較 I 與 J 假如 I > J 則跳到 LBL，如下：

```
     LDA I
     CMP J
     BEQ EQL
     BGE LBL
EQL  ---
     ETC.
```

如果 I = J，則跳到 EQL，而不到 LBJ，若 I ≠ J，則不是小於就是大於，若是大於，則 BGE LBL會發生作用，否則程式繼續作 EQL 部分的指令。

IF/THEN 陳述的模擬
(*IF/THEN STATEMENT SIMULATION.*)

BASIC 中 IF/THEN 陳述的文法如下：

IF <邏輯式子> THEN <陳述>

只有當邏輯式子爲眞時才會執行<陳述>。例如 BASIC 陳述 IF X>=7 THEN Y=0，在 X 大於或等於 7 時會令 Y 等於 0。用組合語言來模擬 IF 陳述時需用各種轉移指令以便跳到所要執行的位置。例如要把上列 BASIC 陳述改成組合語言，則編碼如下：

```
    LDA X
    CMP #$7
    BLT LBL
    LDA #$0
    STA Y
LBL ---
    ETC
```

例 3 中，若 X≧7，則作 BLT 指令下面的指令，讓 Y 等於 0。若 X<7，則 BLT 指令跳過令 Y 等於 0 的指令，直接作 LBL 下面的程式。

可以在轉移指令和目的標示之間放置許多要執行的指令。

FOR/NEXT 迴路 (*FOR/NEXT LOOP REVISITED.*

前面提過，假如可以在值不爲 0 時結束迴路那就更方便一些。現在 CMP 指令就能辦到。假設從索引變數值爲 1 時開始迴路執行一直作到值爲 10 爲止，程式如下：

```
        LDA #$1
        STA I
LOOP    LDA I
        CMP #!10
        BEQ LBL1
        BGE LOPX
LBL1:

;NOTE: The normal code within your loop body goes here

        INC I
        JMP LOOP
LOPX    BRK
        ETC
```

假如間隔是 2 ，則每次在跳到 LOOP 之前把 I 加 2 即可。最後可以再次改良測試步驟。要看 I 是否大於 10 ，就用 BEQ 和 BGE 指令來合成 BGT 轉移，另一種作法是測試 I 是否大於或等於 11 。既然 BGE 是現成的指令。用第二種方法自然程式較短，結果如下：

```
        LDA #$1
        STA I
LOOP    LDA I
        CMP #$B    $B = 11 DECIMAL
        BGE LOOPX

;NORMAL LOOP BODY GOES HERE

        INC I
        JMP LOOP
LOOPX   BRK
        ETC
```

注　　意

到目前爲止我們都只討論不帶符號數字，以後會再討論帶符號數字的比較！

此外轉移指令並不是可以跳到記憶體中任何位置的。轉移指令是用相對位址法，跳開指令後面指明的是 16 位元的絕對位置而轉移指令則是一位元組的分枝（ displacement ）。把分枝加上分歧指令的下一指令的位址就是眞正目的地位址，因此能在轉移指令前126位元組和後129個位元組之間的轉移。

大部分的轉移都在此範圍之內，假如超過此範圍，則編譯器會給你“轉移超過範圍”的錯誤訊息。解決的方法是用相反的轉移指令（例如假如 BEQ 超過範圍則改用 BNE 指令）然後運算元欄字成“ * + $ 5 ”，而在轉移指令之後用 JMP 轉移到目的地的指令。

“ * + $5 ”是什麼意思呢？當6502編譯器碰到“ * ”時就代換成現在指令的起始位址，“ *+$5 ”就是把現在指令的位址加5的意思。因爲轉移指令爲二位元組長而 JMP 指令爲三位元組長，∴“ *+$5 ”就跳到 JMP 指令的下一個指令!

範例：

BEQ LBL 超過範圍，則修改如下：

Simply substitute:

```
BNE *+$5
JMP LBL
```

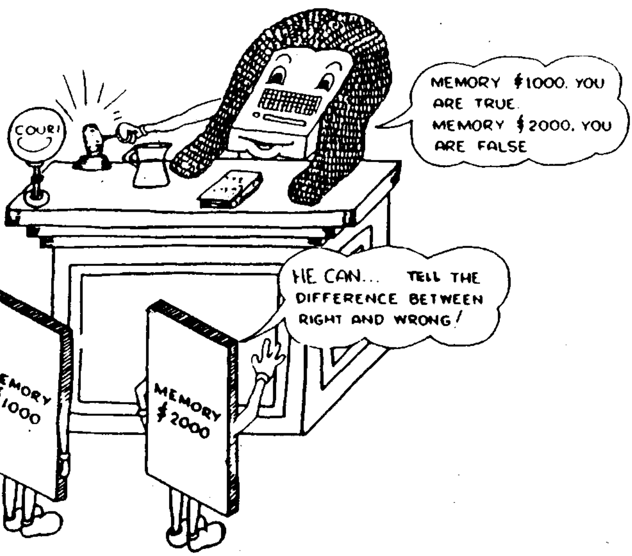
假如上次運算結果使 Z 旗標被設爲 1 ，則會接著作 JMP 指令跳到 LBL ，否則會跳到 JMP 指令的下一個指令去執行，即模擬“假如相等則長跳到 LBL.”！

下表爲轉移和其長跳型式

若轉移超出範圍	用這些指令
BEQ LBL	BNE *+$5 JMP LBL
BNE LBL	BEQ *+$5 JMP LBL
BCC LBL	BCS *+$5 JMP LBL
BCS LBL	BCC *+$5 JMP LBL
BVC LBL	BVS *+$5 JMP LBL
BVS LBL	BVC *+$5 JMP LBL
BMI LBL	BPL *+$5 JMP LBL
BPL LBL	BMI *+$5 JMP LBL
BGE LBL	BLT *+$5 JMP LBL
BLT LBL	BGE *+$5 JMP LBL
BTR LBL	BFL *+$5 (見下節 JMP LBL
BFL LBL	BTR *+$5 JMP LBL

布林值的測試（*TESTING BOOLEAN VALUES*）

在程式中常要用某些變數來保存旗標，以便在其他部分再用，因此要定義布林值。通常用 $00 來代表假而 $01 來代表眞。現在可用 BEQ 和 BNE 指令來測定是眞是假；用 BNE 來測試假的情況，而用 BEQ 來測試眞的情況。聽起來不太對勁，因此 LISA 提供另外二種指令：BTR 和 BFL（branch if true 和 branch if false），這二個指令所產生的碼和 BEQ 及 BNE 的碼相同，只是使用較爲方便而已！



在程式開頭應該有下列二陳述

```
FALSE    EQU $0
TRUE     EQU $1
```

True 和 False 是被看成符號性常數而不是位置，因此

程式變成

```
LDA #FALSE
STA I
LDA #TRUE
STA FLAG
```

來代替下列程式

```
LDA #$0
STA I
LDA #$1
STA FLAG
```

顯然地第一種寫法較易看懂。而符號性常數並不限於用在 true 和 false。每當要用某個有特殊意義的 16 進位數時，就被宣告成符號性的常數（ symbolic constant ），符號性常數會使程式更易看懂且更易修改。

第六章
算術運算

新指令：

ADC SBC

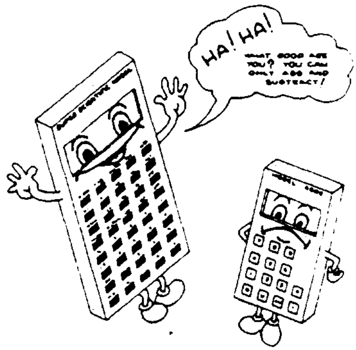
概 論 （*GENERAL*）

6502 有 3 種基本的算術運算指令：(1)不帶符號二元數(2)帶符號二元數(3)不帶符號十進位數。每種都各自有它們的規則，所以不可混合使用。

不帶符號整數(二元數)算術
(*UNSIGNED INTEGER (BINARY) ARITHMETIC*)

6502 所能處理的不帶符號整數的範圍是 0 到 255。雖然這範圍不是頂大，但通常夠用。假如加法的結果超過此範圍，則會被截掉前面部分，只留後面 8 位元，同理當小數減大數時亦發生同樣情形。

但是不必擔心，另有多重精確度運算可用來處理更大的數字。



6502 中沒有 SIN, COS, 1/X, LOG 或 TAN 等多種計算器所有的函數。事實上 6502 連乘法和除法都沒有，6502 只能作加法和減法而其他有用的運算都可用加法和減法來合成！

6502 的加法指令爲 ADC（有進位的加法）。這指令取出某位置的値然後加到累積器上，同時，進位旗標（0或1）也加上去。第一個不帶符號算術的規則是：在作 ADC 之前先清除進位旗標，否則會有 50％之機率會多加了1。

範例：

```
CLC      ;ALWAYS!
LDA #$5
ADC #$3
BRK      ;PRINTS RESULT OF ADDITION
END
```

```
CLC
LDA #7
ADC #$3
BRK
END
```

```
CLC
LDA #$FC
ADC ;#$20
BRK
```

此例中會發生溢位情形，結果前幾位位元被截掉，只留後面8位元。

發生溢位時會如何呢？不像 BASIC 會給你" ** > 255 "的錯誤，而6502 會當作沒事一樣繼續作下去，但結果將是無法預料的。假如在作加法時發生溢位的情形，則進位旗標會被設定，因此用 BCC和 BCS 就可測出溢位的情形了。

範例：

```
CLC
LDA I
ADC J
BCS ERROR ;GO TO ERROR IF OVERFLOW
ETC...    ;OTHERWISE CONTINUE PROCESSING.
```

用進位旗標來通知我們發生溢位情形是很有用的。我們可能忽略不管，或許你確定不會發生溢位，這時候就不必浪費時間或位置來測試溢位了。

當發生溢位時，可保證一件事：結果一定在 $100 到 $1FF 之間。因爲 8 位元所能表示最大數爲 $FF , $FF + $FF 得 $1FE，其他值相加一定小於此值。

溢位的例子：

```
CLC              CLC
LDA #$FF         LDA #$F0
ADC #$1          ADC #$20
BRK              BRK
END              END

CLC              CLC
LDA #$80         LDA #$80
ADC #$80         ADC #$FF
BRK              BRK
END              END
```

不帶符號加法的規則
(RULES FOR UNSIGNED ADDITION)

(1)不要與下面減去或帶符號算術或十進位算術的規則混淆。

(2)在作加法之前，先清除進位旗標。

(3)用 BCS 指令來測試溢位情形，如果發生溢位則進位旗標會被設定

減法 (SUBTRACTION)

減法和加法大致相同，除了三點不同之外：(1)指令為 SBC (有進位的減法)(2)在作減法之前，要設定進位旗標(3)在作完之後若進位旗標為 0，則表示產生不足位情形。

第 2，3 點和加法恰好相反。記住這一點，許多初學者都忘了在作減法之前要先設定進位旗標，以致產生錯誤的結果。

範例： J－I 並把結果放在位置 L

```
         SEC      ;ALWAYS BEFORE A SUBTRACTION!
         LDA J
         SBC I
         STA L
         BCC ERROR
         LDX #$0
         BRK
   ERROR LDX #$FF
         BRK
         END
```

此例中，若一切順利則X暫存器的值應爲$00。若是發生不足位情形，則X暫存器爲$FF。你可以在程式之前再加一些LDA和STA的指令來設定I，J的初值以便作試驗。當然要用假指令 EQU 和 EPZ 來定義I和J的位置。

SBC指令會影響PSW的值，就跟CMP指令一樣。因此可以和CMP指令一樣，在SBC指令之後使用轉移指令，在這裡是不太有用，可是對多重精確的運算卻很有用。

N和V旗標在不帶符號運算中沒有意義。

不帶符號減法的規則
（*RULES FOR UNSIGNED SUBTRACTION*）

(1)不要與加法，有號或十進位算術的規則混淆。

(2)在作減法之前先設定進位旗標。

(3)作完之後若發生不定位，則進位旗標爲0，否則爲1

帶符號算術（*SIGNED ARITHMETIC*）

若作$8減$10時會如何？結果應是$8，可是計算機會告訴你不足位（underflow）（即進位旗標爲0），因爲在不帶符號數字系統中，不允許負數的存在。可是卻常常需要用負數，所以必須發展一套定義有號數字的方法。

6502 是採用2的補數系統來表示負數。在這系統所允許的範圍是－128到＋127之間（用8位元）。有號算術和無號算術作法相同，用指令ADC和SBC，且在加法之前面要先清除進位，在作減法之前要先設定進位旗標。

唯一不同點是進位旗標不再有意義。因爲進位是從位元 7 得來的，但此處位元 7 是符號。眞正發生溢位時，應是從位元 6 進位得來，可是位元 6 的進位並不會影響進位旗標，但卻可用 6502 的 V 旗標來測試溢位或不足位的情況。每當位元 6 產生進位時，這旗標就被設定。

在不帶符號算術中加法和減法中用來測試的指令恰好相反，但在帶符號算術中卻是一樣，溢位或不足位都是用 BVS 來測試。

範例：

```
OVERFLOW OCCURS                        OVERFLOW DOES NOT OCCUR

    LDA #$7F ;127 DECIMAL                  LDA #$1
    ADC #$1  ;1   DECIMAL                  ADC #$2
    BRK      ;RESULT = -128                BRK      ;RESULT = 3
    END                                    END

    CLC                                    CLC
    LDA #$80 ;-128 DECIMAL                 LDA #$FF ;-1 DECIMAL
    ADC #$80 ;-128 DECIMAL                 ADC #$2  ; 2 DECIMAL
    BRK      ;RESULT = 0                   BRK      ;RESULT = 1
    END                                    END

    SEC                                    SEC
    LDA #$80 ;-128 DECIMAL                 LDA #$FF ;-1 DECIMAL
    SBC #$1  ;1    DECIMAL                 SBC #$1  ; 1 DECIMAL
    BRK      ;RESULT = +127                BRK      ;RESULT = -2 ($FE)
    END                                    END
```

TESTING FOR UNDERFLOW/OVERFLOW:

```
    CLC                                    SEC
    LDA #$FF                               LDA #$23
    ADC #$25                               SBC #$43
    BVS ERROR  <- GO IF OVERFLOW ->        BVS ERROR
    BRK        <- STOP OTHERWISE ->        BRK
    END                                    END
```

帶符號算術的規則（*SIGNED ARITHMETIC RULES*

(1)不要與不帶符號算術或十進位算術的規則弄混。

(2)在加法之前要先清途進位旗標，而在作減法之前要先設定進位旗標。

(3)用 BVS/BVC 指令來測試溢位或不足位情形（當 V = 1 時表示發生溢位或不足位）

帶符號的比較（*SIGNED COMPARISONS*）

利用 V，N 和 Z 旗標可作有號比較。當二數相等時，Z 旗標會爲 1，所以可用 BEQ/BNE 指令來測試相等的情形。當 V 旗標等於 N 旗標時，即表示累積器的帶符號值大於等於記憶位置的值，否則則小於記憶體位置的值。

目前只剩兩個問題，首先沒有測試 V 旗標和 N 旗標的指令，其次 6502 的 CMP 指令不會改變 V 旗標的值。

第二個問題較易解決，可用 SBC 指令。SBC 指令對旗標的影響和 CMP 指令一樣，除了一點，它會改變，V 旗標的值。因此帶符號比較的作法如下：先設定進位旗標（在減法之前一定要作的），然後用 SBC 指令來代替 CMP 指令。

第一個問題比較麻煩，下列程式即用來模擬帶符號 BGE 指令和帶符號 BLT 指令。

```
SEC
LDA  A
```

```
     SBC B
     BMI LBL
     BVC GE
LT:            <- BRANCH TO HERE IF A < B



LBL  BVC LT
GE:            <- BRANCH TO HERE IF A >= B
```

二元碼十進位算術

(*BINARY CODED DECIMAL ARITHMETIC*)

現在要介紹另一個數字系統，叫作二元碼十進位系統或簡寫成 BCD 。這種系統在輸出入使用最方便. BCD 是一種以二元碼來代替十進位數的方法，其型式如下：

DECIMAL DIGIT	BINARY REP.	BCD REP
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	0101
6	0110	0110
7	0111	0111
8	1000	1000
9	1001	1001

DECIMAL DIGIT	BINARY REP.	BCD REP
10	1010	0001 0000
11	1011	0001 0001
12	1100	0001 0010
13	1101	0001 0011
14	1110	0001 0100
15	1111	0001 0101
16	0001 0000	0001 0110

從 1 到 9 ， BCD 和二進位表示法都一樣，但從 10 開始

就不同了。在 BCD 中，低階尼波用來表示十進位中的低位數，而高階尼波則表示高位數。在 BCD 中不允許 1010 到 1111 之間的數。用 8 位元可表示 0 到 99 之間的數值，在組合語言中說明 BCD 的方式和說明 10 進位數的方式一樣，只是BCD 常數之前是加個“ $ ”。

不帶符號的BCD算術
(*UNSIGNED BCD ARITHMETIC*)

跟帶符號或不帶符號的二進位算術一樣，這裡的加法和減法都是用 ADC 和 SBC 指令執行的，同時在加法之前也要清除進位而在減法之前也要設定進位。在作完加法之後若發生溢位則進位旗標會爲 1 ，而作完減法時若發生不足位，則進位旗標會爲 0 。

既然無號的 BCD 運算和無號的二進位運算的指令一樣，處理機一定得有某種方法來知道是該執行十進位或二進位的運算。這是透過一個可用程式控制的旗標（ D，或十進位旗標）來達成的。若 D = 1 則作十進位算術，否則作二進位算術。可用指令 SED 和 CLD 來設定或清除 0 旗標。

一旦設定 D 旗標，則所有算術都是十進位，直到再清除 D 旗標爲止。但這也有個缺點，若是設定之後忘了清除，則所有想作的二進位算術都變成無效了。因此要小心使用 D 旗標。因此除非馬上就要作十進位算術，否則在程式一開始最好先用 CLD 指令，以免不必要的錯誤。

另外還要注意二件事，第一件是若運算元的任一個尼波的值在 1010 到 1111 之間，則此運算無效，第二是 6502 本身的錯誤，你必須在加法之後用 SBC 之指令來檢查累積器

是否為 0 。

十進位算術範例 (DECIMAL ARITHMETIC EXAMPLES:)

```
SED         ;SET DECIMAL MODE
CLC         ;ALWAYS BEFORE AN ADDITION
LDA #$25    ;INITIALIZE ACC TO 25 (DECIMAL/BCD)
ADC #$10    ;ADD 10 (DECIMAL/BCD)
BRK         ;RESULT IS 35
END

SED
SEC         ;ALWAYS BEFORE A SUBTRACTION
LDA #$52    ;INIT TO DECIMAL 52
SBC #$22    ;SUBTRACT 22 (DECIMAL/BCD)
BRK         ;RESULT IS 30
END

SED
CLC         ;ALWAYS BEFORE AN ADDITION
LDA #$99    ;LOAD WITH 99 (DECIMAL/BCD)
ADC #$1     ;ADD 1 (DECIMAL/BCD)
BRK         ;RESULT IS 0, CARRY = 1
END

SED
SEC
LDA #$00
SBC #$1
BRK         ;RESULT IS 99, CARRY = 0;
END
```

不帶符號算術的規則
(*UNSIGNED ARITHMETIC RULES*)

(1)用 SED 指令來設定十進位旗標。

(2)在作加法之前先清除進位，而在減法之前先設定進位。

(3)要先確定運算元為有效的 BCD 數字，否則會產生錯誤。

(4)假如作加法之後，要測試 Z 旗標，則要加個指令"CMP　#$00"。

(5)加法的溢位是由進位旗標為 1 來顯示的，在這種情形下會產生大於 99 的值。

(6)若 C = 0 則表發生減法的不足位，這時所產生的值小於 00。

帶符號的BCD算術（*SIGNED BCD ARITHMETIC*）

6502 並沒有提供帶符號的十進位算術。

摘要　（*ARITHMETIC REVIEW*）

這章中我們介紹了三種數字系統：不帶符號的二進位，帶符號的二進位和 BCD。

BCD 在作 I/O 動作和某些較少被使用的數值計算時特別方便。許多儀器，像伏特計，頻率計算器和鐘的輸出都是 BCD 資料。假如要將 APPLE II 和這些儀器接起來，則要採用 BCD 系統。

不帶符號的二進位數則在只有正數的情形時才被使用，約 95 % 是這種情形。只有當要處理負數時才用帶符號的算術。因為不帶符號的算術較快，較易作，所以比其他二種都更有用。

8位元算術的規則
(RULES FOR 8-BIT ARITHMETIC)

(1)最大值

(a)帶符號＝ 127
(b)不帶符號＝ 255
(c)十進位＝ 99

(2)最小值

(a)帶符號＝－ 128
(b)不帶符號＝ 0
(c)十進位＝ 0

(3)假如需要，程式員必須自己提供測試溢位和不足位情形的副程式。

(4)在加法之前先清除進位，而作減法之前要先設定進位旗標。

(5)所有的運算都是經過累積器來作的。

8位元算術有些限制，最重要的一點是範圍的限制。以後再討論多重精確度的算術。

Retro Worksho,
果粉工作室

第七章

副程式和堆疊處理

新指令：

PHA	PLA	JSR
PHP	PLP	RTS

概　論（GENERAL）

跟 BASIC 一樣，組合語言程式員也常常需要跳到一組指令去執行，然後再跳回來作下一個指令。

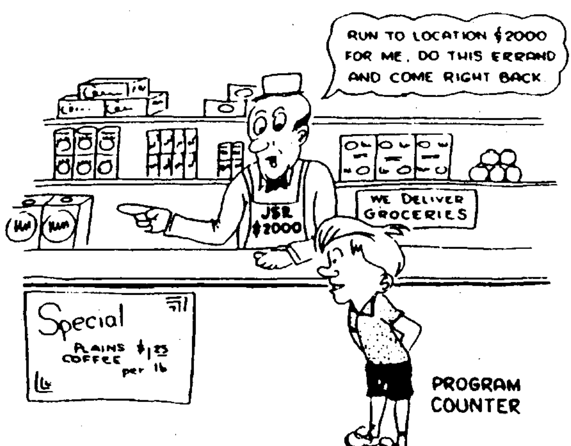
這種技巧稱爲副程式。在 BASIC 中用 GOSUB 陳述就可跳到副程式中去作。當作完之後再用RETURN 回到主程式來。

組合語言的作法類似，用 JSR 指令 (Jump to Subroutine) 跳到副程式，再以 RTS (Return from Subroutine) 指令跳回主程式。JSR 指令與 JMP 類似，都是一位元組的指令碼再加上二位元組的絕對位址。而 RTS 則是一位元組長的指令。

在 BASIC 中副程式通常被使用來設定初值或避免重覆碼，但在組合程式中有更重要的原因需要用副程式—利用副

程式把複雜的工作分成幾個較小而簡單的次工作。前面提過 6502 只能作加減法，假如要作乘法時怎麼辦呢？用副程式來作。在 BASIC 中你不必用副程式來作乘法，因爲乘法在其本身語言中就有，但在組合語言中就沒有，因此必須用副程式來模擬。此外 I/O 也需要用副程式，因爲 6502 並沒有提供"PRINT" 或 'INPUT' 的陳述，必須用副程式來模擬。

副程式最重要的用處是可以把工作分段成較易編碼的小工作。這種發展方法稱爲"從上往下"（"TOP-DOWN"）的方式。你可能常聽到"結構化程式"這個名詞。結構化程式必須用到如 FOR 迴路， REPEAT/UNTIL 迴路， WHILE ， IF/THEN/ELSE 陳述和 BEGIN/END 等這些程式架構。顯然地在組合語言中結構化程式不可能，因爲它沒有前面所說的陳述。雖然你可以用 JMP 的指令來模擬它們，但結構化程式是不允許用 GOTO（或 JMP）的，因此結構化程式設計在組合語言上是不可能的。然而模擬這些陳述（IF/THEN/ELSE, REPEAT/UNTIL 等）卻是個不錯的構想，以後將會討論！



往上往下程式設計的觀念如下：先定問題大綱，不要牽涉細節，其次再把每個綱要分成小段，考慮較細節的部分。接著再把這些小段再分，直到可以用組合語言很容易地實行為止。

在定義問題的每一步驟就應對應於一個副程式。主程式只有一些設定初值，測試，和 JSR 的指令， JSR 就是跳到許多處理細節的副程式。同理這些副程式也只是一些設定初值，測試，某些資料處理和 JSR 指令的集合。

最後工作會分割到最低階層，其中副程式只由不是 JSR 的指令所組成。自然副程式的深度視實際應用而定。

平常的組合語言程式非常難偵測，用這種方式寫的程式則較易偵錯，如果不用"從上往下"法寫程式可能較快，但偵錯卻要花很多時間。

變數問題（*VARIABLE PROBLEMS*）

組合語言的副程式和 BASIC 一樣有許多問題。例如下面的 BASIC 程式：

```
10 FOR I=1 TO 10
20 GOSUB 50
30 NEXT I
40 END
50 I=1
60 RETURN
```

例子中 FOR/NEXT 迴路應該執行 10 次。可是第 50 行的呼叫副程式會把 I 設定為 1 。因此這迴路會不停地作下去。在組合語言中也有這種問題。或許你會說只要在副程式中

不要用到迴路控制參數就可以了！在組合語言中，可以和 BASIC 一樣用不同的變數名稱，可是暫存器卻不行，如下例：

```
       LDA #$F
LBL    JSR SETX
       DEX
       BNE LBL
       BRK
.
.
SETX   LDX #$10
       RTS
```

上例中，X 暫存器用來作迴路控制參數。副程式 SETX 把 $10（即 16 ）的值存入 X 暫存器中，然後再跳回。其後 X 暫存器的值被減 1（變成 $F 或 15 ），成爲不是零的值。因此這廻路會一再反覆！

當然無法替 X 暫存器改名。但不允許程式員使用 X 暫存器（或累積器或 Y 暫存器）卻不合理，不准在副程式中使用 X 暫存器倒不如在進入副程式時把 6502 相關的暫存器的值保存起來，到跳回主程式時再還原成原來的值。上例因此可改寫如下：

```
XSAVE   EPZ $0      ;SAVE LOCATION FOR THE X REGISTER
        LDX #$F
LBL     JSR SETX
        DEX
        BNE LBL
        BRK
.
.
SETX    STX XSAVE
        LDX #$10
        LDX XSAVE
        RTS
        END
```

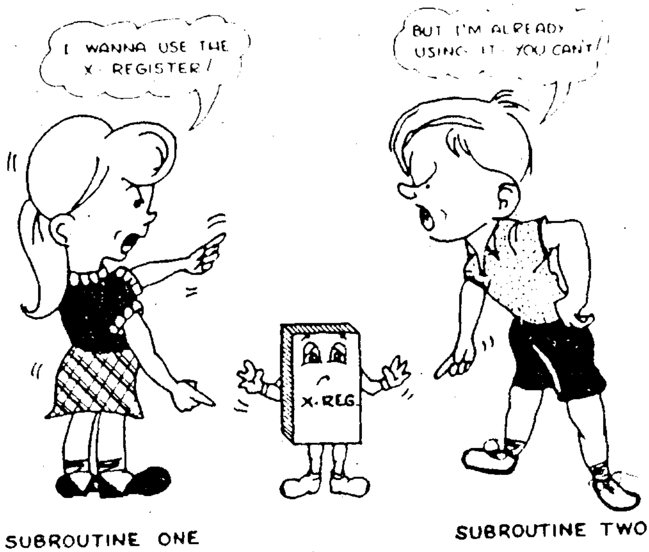
這樣子並沒有解決副程式和暫存器用法的所有問題。看

看下面的碼：

```
XSAVE    EPZ $0
         LDX #$F
LBL      JSR SETX
         DEX
         BNE LBL
         BRK
:
:
SETX     STX XSAVE
         LDX #$10
         JSR SETX2
         LDX XSAVE
         RTS
:
:
SETX2    STX XSAVE
         INX
         LDX XSAVE
         RTS
         END
```

首先是建一個廻路，在廻路中呼叫副程式 SETX ，爲了避免發生不停的廻路循環，首先把X暫存器的值存在 XSAVE 位置上。接著X暫存器中載入值 $10，然後再呼叫 SETX2 。同樣地要再把X暫存器的值存起來，免得 SETX2 破壞了原來的值。問題是如此一來X暫存器的現值會把前一個X暫存器的值（存在 XSAVE 位置上）改掉，因爲二者都存在 XSAVE 位置上。因此當從 SETX2 跳回 SETX 時，X暫存器載入 $10，跟呼叫 SETX2 之前一樣，但當要從 SETX 跳回主程式時，就發生錯誤了，此時XSAVE 的值爲 $10，將其載入X暫存器中，則X暫存器的值與呼叫 SETX 之間的“$F ”值不同，因此這個程式又有不停的廻路！

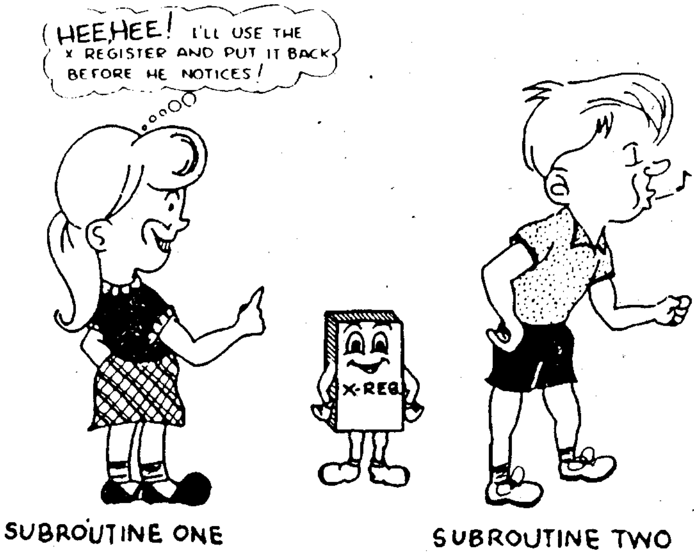
解決的方法是每個副程式用不同的變數（那位址）來存暫存器的值。



範例：

```
XSAVE1   EPZ $0
XSAVE2   EPZ $1
;
         LDX #$F
LBL      JSR SETX
         DEX
         BNE LBL
         BRK
;
;
SETX     STX XSAVE1
         LDX #$10
         JSR SETX2
         LDX XSAVE1
         RTS
;
;
SETX2    STX XSAVE2
         INX
         LDX XSAVE2
         RTS
         END
```

這個程式就不會產生不停的廻路。讓每個副程式有不同的變數名詞時，可用 DFS（ define storage ）在副程式一開始時就保留一個位元組。



7–7

範例：

```
         LDX #$F
LBL      JSR SETX
         DEX
         BNE LBL
         BRK
:
:
XSAVE1   DFS 1      ;RESERVE ONE BYTE FOR THE X REGISTER
:
SETX     STX XSAVE1
         LDX #$10
         JSR SETX2
         LDX XSAVE1
         RTS
```

```
XSAVE2   DFS 1
:
SETX2    STX XSAVE2
         INX
         LDX XSAVE2
         RTS
         END
```

只被一個副程式使用的變數稱爲該副程式的局部變數(Local variables),應該在副程式用到它之前就先定義好,避免混淆。而被多個副程式共用的變數稱做共用變數(GLOBAL variables)。對 APPLE II 的組合語言應用程式來說,可用局部變數來保存暫存器的值。

有二種副程式不能有局部變數,一爲"再進入"("REENTRANT")副程式,一爲可存在 ROM 上的副程式。因爲再進入副程式可以呼叫自己,如果採用局部變數,則暫存器的值一定會被破壞。

可存在 ROM 的副程式無法用 DFS 指令來保留位置,因爲保留給它的記憶位置是在 ROM 上。雖然可以用 EQU 來定義 RAM 上的位置,可是卻不方便。

假如可以存值在某一奇異的位置,等要存下一個值時就把前一個值移到另一個位置,如此就不會蓋掉原來的值。當把這奇異位置的值載入暫存器中則會取得最後存入該位置的值,而再把前一個存入的值放回這奇異的位置。如此我們就可以寫下列程式:

```
         LDX #$F
LBL      JSR SETX
         DEX
         BNE LBL
         BRK
```

```
SETX     STX "MAGIC"
         LDX #$10
         JSR SETX2
         LDX "MAGIC"
         RTS
  :
  :
 ETX2    STX "MAGIC"
         INX
         LDX "MAGIC"
         RTS
         END
```

在上例中我們把值 $F 載入 X 暫存器然後呼叫 SETX，在 SETX 中把 X 暫存器的值存入奇異的位置。再把 $10 載入 X 暫存器，再呼叫 SETX2 。此時又把 X 暫存器的值存入奇異的位置，這會使得前一個值被移到其他地方（自動地移動）。接著把 X 暫存器加 1 得 $11，下一個指令把奇異的位置的值 $10 載入 X 暫存器中，因此最先存在奇異的位置的值 $F 又存回原來的位置。這時 SETX 跳回呼叫程序 SETX 再把奇異的位置上的值（現在是 $F ）載入 X 暫存器中，也跳回呼叫程序，這時 X 暫存器的值和呼叫 SETX 之前的值完全一樣！

這奇異的位置是 LIFO （後進先出 (Last In, First Out) ）資料結構的例子，通常稱爲堆疊。

6502 微處理機提供了 LIFO 堆疊。可以用 PHA 指令把累積器的值壓入堆疊中，而用 PLA 指令把堆疊的值拉出放入累積器中。因此 PHA 就是把累積器的值存入奇異的位置的方法，而 PHA 則是把奇異位置上的值載入累積器的方法。

假如你把累積器的值壓入堆疊中，卻不拉出來，則只是把它放在堆疊頂端而已，可是如果你沒先把資料壓入堆疊就要將其拉出，則會發生問題！ 6502 的堆疊會隨著叫作堆疊指標的一個 8 位元暫存器來旋轉。這指標的值永遠在 $100

和$1FF之間。當把資料壓入堆疊時，資料存在指標指在第一頁位置上，接著把指標減1，因此下一個被壓入的資料會存在前一個壓入的資料之下。堆疊指標永遠指到下一個可用的位置。當要拉出資料時則指標加1，再把指標所指位置的值載入累積器中。除非已經壓入超過256位元組的資料到堆疊中。否則每次使用 PHA 都保證會把值存在新的且唯一的位置內，因爲成標是8位元，頂多能壓入256個位元組的資料到堆疊中，如果超過，256個則堆疊會轉回第一個位置，如此一來就會蓋掉以前的資料。大致上來說，256夠多了，通常程式頂多用到64個位元組的暫時位置就夠了。

沒有特別給X，Y暫存器的壓入和拉出指令。要把這些暫存器壓入堆疊中，可以先把值轉入累積器中再用 PHA 推入堆疊中。

範例：

Y暫存器壓入	把X暫存器壓入
TYA	TXA
PHA	PHA

需注意這會改掉累積器中原來的值。

現在我們用PHA和PLA在副程式中保存暫存器。如下:

```
        LDX #$F       ;INIT INDEX COUNT
LBL     JSR SETX
        DEX           ;DECREMENT COUNT
        BNE LBL       ;LOOP IF NOT THROUGH
        BRK           ;STOP


SETX    PHA           ;SAVE THE ACCUMULATOR
        TXA           ;SAVE THE X REGISTER
        PHA
        TYA           ;SAVE THE Y REGISTER
        PHA
```

```
        LDX #$10
        JSR SETX2
;
        TXA             ;NOW RESTORE THE Y REGISTER
        TAY
        PLA             ;RESTORE THE X REGISTER
        TAX
        PLA             ;RESTORE THE ACCUMULATOR
        RTS
;
SETX2   PHA             ;SAVE ACC
        TXA             ;SAVE X REG
        PHA
        TYA             ;SAVE Y REG
        PHA
;
        INX
;

;RESTORE THE Y REG


;RESTORE X REG
;RESTORE ACC
```

或許你已注意到暫存器被拉出的順序恰好與被壓入的順序相反，因爲堆疊是 LIFO（後進先出）的資料結構。

而 6502 如何記住作完副程式後要跳回那個位址呢？當執行指令 JSR 時，跳回位址就被壓入堆疊中，而執行 RTS 時把跳回位址從堆疊中拉出。假如在執行 RTS 之前沒有把壓入的資料都拉出來，則資料會被當作跳回位址使用。因此記住一個規則：在執行 RTS 之前要把所有壓入堆疊的資料拉出。但也不要拉出過多的資料，否則會把跳回位址弄掉。

這會促成從上往下程式設計中非常重要的一點：副程式中應該只有一入口和一出口。假如有許多出口，則很可能會忘記拉出堆疊中所有的資料。解決的方法很簡單，不用很多的 PTS，而用 JMP，跳到唯一的 RTS 上去。

範例：

```
SUBRT PHA
;
      LDA LOC1
      CMP #$50
      BLT SUB1
;
      LDA #$0
      STA LOC1
      JMP SUBX
;
SUB1  INC LOC1
SUBX  PLA
      RTS
```

在副程式中沒有用到X，Y暫存器，因此不必存到堆疊中。

當碼為重覆時，副程式就很有用，因此就將把暫存器壓入堆疊，和將其值從堆疊拉出的動作寫成一副程式，叫作 SAVE 和 RESTR，經驗不夠的程式員可能編碼如下：

```
SUBRT JSR SAVE

   -SUBROUTINE-
   -CODE  GOES-
   -   HERE   -

      JSR RESTOR
      RTS
;
;
SAVE  PHA
      TYA
      PHA
      TXA
      PHA
      RTS
;
;
RESTR PLA
      TAX
      PLA
      TAY
      PLA
      RTS
      END
```

避免用這個程式。把累積器X，Y暫存器壓入堆疊中，接著就做 RTS，則6502會把最後用入堆疊的二位元組看成跳回位址，但X，Y暫存器的值根本不是跳回位址。

傳送參數（*PASSING PARAMETERS*）

參數是用來傳送資料到副程式的。例如 sin (X) 中，X 就是函數 SIN 的參數。 SIN(X) 會送回 X的正弦函數值。POKE也是一個有參數的程序（ procedure ）。事實上 POKE 有二個參數：第一個參數爲第二個參數要存放的位置。

在 6502 中有很多方法可用來傳送參數給副程式。最簡單的方法是用暫存器來傳送。它有個缺點，參數只能有三位元組，對某些應用來說也許夠了，例如APPLE監督程式 ROM 中有兩個副程式，可把累積器的值以 ASCII 字元印在螢幕上及從鍵盤上讀入字元轉換成 ASCII 存在累積器中。兩個副程式分別放在位址 $FDED 和 $FD0C 上。你可以用下列程式把 APPLE II 變成一部電動打字機：

```
COUT   EQU $FDED        ;USE A SYMBOLIC LABEL FOR
                        ;CHARACTER OUTPUT

RDKEY  EQU $FD0C        ;SAME FOR KEYIN ROUTINE

LOOP   JSR RDKEY
       JSR COUT
       JMP LOOP
       END
```

要停止程式的執行，打 reset 鍵就可以。

當你需要傳送多於三位元組的參數時，就要用另外的方

法。一種方法為把參數放在某些已知的位置，然後在副程式中，再到這些已知位置拿資料。同樣地在執行完副程式之後可以到某一已知位置拿出送回的資料。例如下列程式（SUM），求 4 位元組的總和：

```
        LDA I
        STA PARM1
        LDA J
        STA PARM2
        LDA K
        STA PARM3
        LDA L
        STA PARM4
        JSR SUM
        LDA RESULT
        BRK
PARM1   DFS 1
PARM2   DFS 1
PARM3   DFS 1
PARM4   DFS 1
RESULT  DFS 1
SUM     CLC
        LDA PARM1
        ADC PARM2
        CLC
        ADC PARM3
        CLC
        ADC PARM4
        STA RESULT
        RTS
        END
```

假如需要也可以測試是否有溢位！和用暫存器一樣，參數應該是局部變數，其他副程式不可使用！

Retro Workshop
果粉工作室

第八章

陣列第零頁索引和間接位址法

新指令：

HEX ORG OBJ DFS ASC

概 論（GENERAL）

到目前爲止我們只用過絕對（ 16 位元位址），直接（ 8 位元資料）和相對（ 8 位元轉移位址法。這些是最常用的但並不是唯一可用的。

第零頁位址法 （ZERO PAGE ADDRESSING）

6502 的 64K 位址空間被分成 256 個具有 256 位元組的單位。每一個 256 個位元組稱爲一頁（pages），從 0 算到 $FF。第 1 頁留作 6502 的堆疊，而第 0 頁通常用來存變數和指標。在 Apple 監督程式，DOS 和大部分像 BASIC 和 Pascal 的語言中，第零頁常被用到。假如你要用這些語言（或從組合語言程式中用到 DOS ），則你必須特別小心第

0 頁的位置。假如你要用第 0 頁中某一個已被主要語言所用的位置，則會發生“第 0 頁衝突”，而你的程式就會產生錯誤。爲了避免錯誤你應該去查看新的 Apple 手册中第 74 到 75 頁的第 0 頁位置表，了解那些位置已被某種高階語言所使用了。

既然會發生衝突，爲什麼要用第 0 頁的位置呢？還有其他 48496個位置可用啊！因爲 6502 提供了“第 0 頁位址法”。第 0 頁位址法的指令由一位元組的指令碼和一位元組的位址所組成。因爲 8 位元只能指明 256 個不同的位置，因此只能參考到記憶體中的一頁：就是第零頁。這種指令只要二位元組，所以可以節省記憶空間。另一個好處是這指令較絕對位址法的指令執行的速度快。（事實上快了¼）。

當下列情況時，會自動採用第零頁位址法：

(1)所指的位址爲非符號性的（也就是說不用標示）而且值小於 $100。

例如：

```
LDA $1
STA $FF
LDX $1
ADC $25
```

(2)位址爲符號性的，且用“EPZ”宣告。

例如：

```
LBL       EPZ $0
LBLA      EQU $0

          LDA LBL          ;ZERO PAGE ADDRESSING USED
          LDA LBLA         ;ABSOLUTE ADDRESSING USED
```

記住只有當標示使用 EPZ 來定義時才用第零頁位址法否則都用絕對位址法。

組合語言中的陣列
（*ARRAYS IN ASSEMBLY LANGUAGE*）

有時候會需要用到陣列或字串。陣列是一些長度相同的元素所組成，且存在記憶體中連續的位置，所以可用對第一個元素位置的分枝來取出任何一個元素。

首先考慮在組合語言中如何來定義陣列？有許多種方法，基本上要先決定保留那塊位置來存陣列。因此可以用EQU假指令來宣告陣列。例如假設要保留 40 個位元組，其次要決定要放在記憶體那裡，當然需確定不會放在你的程式碼或如 DOS 碼的位置上。第 3 頁是放小型陣列的好地方。因此用下列指令定義陣列從位置＄300 開始存放：

```
ARRAY EQU $300 ;ARRAY IS $28 (40) BYTES
LONG
```

這指令說明陣列 ARRAY 從位置＄300開始，程式員必須自己記住位置＄300 到＄327 是給陣列 ARRAY 用的。假如你還要宣告另一陣列，佔用 10 個位元組，則如下：

```
ARRAY EQU $300 ;ARRAY IS $28 (40) BYTES
LONG
ARRAY2 EQU $328 ;ARRAY IS $A (10) BYTES
LONG
```

這樣子就保證 ARRAY2 的空間和 ARRAY 的空間不會發生衝突了。

顯然地自己計算位置是很麻煩的，因此更好的宣告方式

是：

```
ARRAY    EQU $300
ARRAY2   EQU ARRAY+$28
;
;ARRAY2'S LENGTH IS $A
;
```

這指令說明 ARRAY2 是從 ARRAY 開頭的後 40 個位置開始。在 ARRAY2 後面的註解只是說明 ARRAY2 有多長，假如要再加其它陣列則可決定該如何宣告。

用 EQU 指令有兩個缺點。第一個是當你寫程式時需知道陣列存放在那裡，這會形成編碼的不夠效率，尤其當陣列很大時，無法知道程式何時才結束（通常可把陣列放在程式的後面，便成爲一整塊位置）。第二個缺點是必須記住最後一陣列的長度，如此才能再加上其他的陣列。

假指令 ORG 會把位置計算器的值設定爲運算元欄中所指定的位址。位置計算器是用來決定現在的碼該放在那裡的指標。通常編譯程式時，會自動從位置$800 開始存放。每當建立一位元組的碼。就存在位置計算器所指的位置內，然後位置計算器再加 1 。每次碰到標示時就把標示和其定義的位置計算器的值存入符號表中。看下列程式：

```
        ORG $800        ;DEFAULT VALUE
        JMP LABEL       ;THREE BYTE INSTRUCTION
ARRAY   ORG $903
LABEL   ---             ;THE REST OF YOUS PGM GOES HERE
```

此例中編譯器被指示由位置$800（即等於內定值）開始存程式，在 ORG 之後是 JMP 指令，因爲 JMP 指令爲三位元組長，因此下一個指令從位$803 開始存。再下一個指令包含一個標示（“ ARRAY ”），所以這標示就與現在位置

計算器的值被存入符號表中。注意，除了 EQU 和 EPZ 之外，標示都在這一行指令碼產生之前被存入符號表中。因此"ARRAY"就與值 $803 被存入符號表中了。接著 ORG指令強迫令位置計算器值爲 $903 ，即保留了 256 個位元組給程式中的陣列使用。這個方法唯一的缺點是必需知道目前位置計算器的值，但常常不可能知道，因此無法用 ORG 來保留位置。

當編譯器在運算元欄中碰到" * "時，會把" * "換成目前位置計算器的值，如此就不用猜現在位置計算器的值，而以" * "來代替就可以了。現在保留 256 位元組的方法如下：

```
        ORG $800
        JMP START
ARRAY   ORG *+$100      ;RESERVE 256 LOCATIONS
START   ---             ;CODE GOES HERE
```

另一個問題是"程式和資料的分開"。假如你把陣列放在程式中，必須保證不會被當作碼來執行，否則會發生錯誤。在上例中， JMP 指令使程式執行跳過陣列部分。通常存在程式中的陣列和其他資料只能放在改變程式流程的指令如 JMP ， RTS 和 BRK 之後。

另一個問題是用假指令 ORG 和用假指令 OBJ 。什麼是假指令 OBJ 呢？ LISA 在翻譯程式時會用到除了 $800 到 $1800 範圍之間以外的任何位置。假如你希望你的程式從位置 $4000 開始執行時怎麼辦？也不行從位置 $800 開始編譯程式，完了之後再把碼移到位置 $4000 來執行！大部份 6502 組合語言是不能重新放置的（ relocatable , relocatable 表示程式可以在記憶體任何位置執行而不發生問題）。不幸地

地是如 JMP， JSR 等指令都是指明絕對位置。如編譯下列程式

```
        ORG $800
        JMP LBL
ARRAY   ORG *+$100
LBL     LDA ARRAY
        STA ARRAY+$1
        BRK
        END
```

然後再移到位置$4000去執行，則會發生錯誤，因爲ORG把程式位置定爲$800，所有的絕對位址都是相對於這個位置。如 ARRAY 的位址應是$803 而 LBL 則爲$903 。當編譯程式時第一個 JMP 指令被換成 JMP $903 。假如把程式移到位置$4000，則第一個指令就會又跳回原來的位址（ $903 ）來執行了！

這是不是就表示所有的組合語言都必須存放在位置$800到$2000之間呢？當然不是，只是當翻譯程式時，目的碼會被放在這個範圍內。當碰到假指令 ORG 時，碼計算器（指到碼應存放的位置的指標）就改成運算元欄，所指定的值。例如程式中有 ORG $4000則表示程式會被存在$4000位置上，並且在此位置上被執行。

假如你想讓你的程式在$800 位置上編譯而在$4000位置上執行，則可以使用下列程式碼：

```
        ORG $4000
        OBJ $800
        LDA #$0
        STA LBL
        BRK
LBL     EQU $0
        END
```

假指令 OBJ 會把碼計算器的值設定爲其運算元欄所指定的位址。這與陣列宣告又有什麼關係呢？看下面的程式：

```
        ORG $4000
        OBJ $800
        JMP LBL
ARRAY   ORG *+$100
LBL     LDA ARRAY
        STA ARRAY+$1
        BRK
        END
```

ORG $4000 指令保證會產生正確的碼而 OBJ $800 則保證目的碼放在記憶體 $800 到 $1800 的範圍內。 JMP 指令則保證資料不會被當作程式來執行。但 ORG*+$100，用來替陣列保留位置則有問題。記住 ORG 指令會改掉位置計算器和碼計算器的值。這表示當碰到" "ORG *+$100" "時，位置計算器的值是 $4103（如我們所要），而碼計算器的值也是 $4103(卻不是我們所要的)。這時再多加指令 "OBJ *+$100" 也於事無補，因爲在執行 OBJ *+$100 時，位置計算器的值已經是 $4103 了！這問題可以解決，可是 LISA 提供了另一種作法。

LISA 提供了另一個指令 DFS (define storage)，來處理這個問題。當碰到 DFS 指令時， LISA 會把程式計算器和碼計算器的值加上 DFS 指令中運算元欄所指定的位元組數目。在上例中要保留 256 個位元組可以寫成：

```
        ORG $4000
        OBJ $800
        JMP LBL
ARRAY   DFS $100
LBL     LDA ARRAY
        STA ARRAY+$1
        BRK
        END
```

DFS 指令會自動保留位置給你。用 DFS 時也要將陣列存在不會被執行的位置內。 DFS 可以用來定義單變數或陣列，如 DFS $1 。

在編譯時給定陣列初值
(*INITIALIZING ARRAYS AT ASSEMBLY TIME*)

有時候要在編譯時給定陣列初值，例如要把資料表存入記憶體中或給定某些字串等。到目前爲止，尚沒有方法可行。通常存在陣列中的資料有二種：數字或字串。可用假指令 HEX 來將 16 進位的值存入記憶體中，尤其當建立表時特別有用。 HEX 可在程式中如下使用：

```
        JMP LBL
ARRAY   HEX 00010203
LBL     LDA ARRAY
        STA $0
        BRK
        END
```

例子中累積器被載入位置" ARRAY "所存的值。下一個指令把累積器的值存入位置 $0 中。 HEX 指令所要的是二個 16 進位數所組成的項目，否則會產生錯誤。這些值都存在記憶中連續的位置內。

使用假指令 ASC 來設定文字初值。如下：

```
          JMP LBL
ARRAY     ASC "HI THERE"
LBL       LDA ARRAY
          JSR $FDED
          LDA ARRAY+1
          JSR $FDED
          LDA ARRAY+2
          JSR $FDED
          LDA ARRAY+3
          JSR $FDED
          LDA ARRAY+4
          JSR $FDED
          LDA ARRAY+5
          JSR $FDED
          LDA ARRAY+6
          JSR $FDED
          LDA ARRAY+7
          JSR $FDED
          BRK
          END
```

這個程式會在螢幕上打出 "HI THERE" 。在 ASC 後面的 ASCII 字串要以單引號或雙引號括起來。從現在開始我們都用雙引號，待會再討論單引號，待會再討論單引號的用法。

假如在字串中有雙引號時，怎麼辦？只要用二個雙引號來表示一個雙引號即可。

範例：

```
ASC "HOW'S ""THIS"""
```

第一個雙引號爲劃界符號，既然" 爲劃界符號，所以單引號就可出現在字串中。在字串中要包括雙引號則連續用二個雙引號來代替，這會告訴 LISA ，這個引號不是劃界符號而是字串的一部分。這例子會在記憶體中產生下列字母：

```
HOW'S "THIS"
```

現在你可以保留位置給陣列，那麼要如何處理陣列元素呢？要處理單一元素很簡單，只要把第一個元素的位置再加上分枝就可以。例如要取陣列 ARRAY 的第 10 個元素則以 ARRAY + $9 爲位址即可。（記住，組合語言中陣列的索引是從 0 開始）。這是固定分枝，也就是說在執行時保持固定，只要在翻譯時算出即可。假如下列指令：

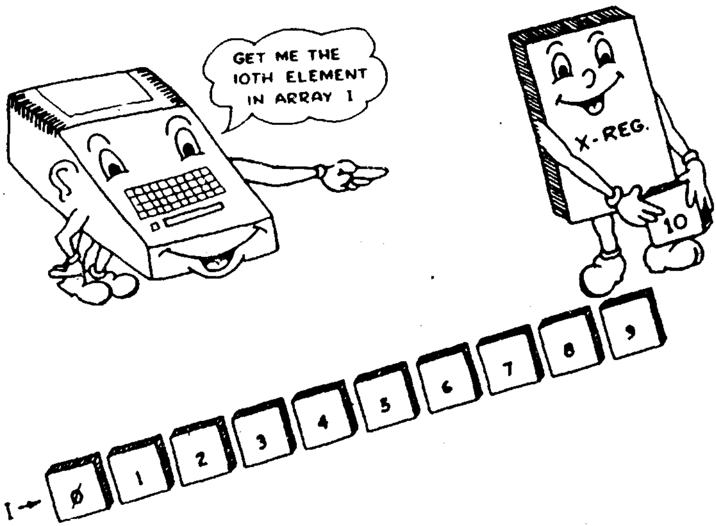
```
LDA  ARRAY+I
```

則在翻譯時會把 I 的位址加上 ARRAY 的位置作爲要載入累積器的值所在的位址，也就是說在翻譯時就已決定要載入那個位址內的值了。例如 I 被存在位置 $1000 而 ARRAY 存在位置 $2000，則 LDA ARRAY+I 會把位置 $3000 上的值載入累積器中，這個 $3000 是在翻譯時就計算好了的。因此要如何模擬高階語言中使用變數作爲陣列索引的特性呢？請繼續看下去！

用索引位址法來處理陣列元素
(*USING INDEX REGISTERS TO ACCESS ARRAY ELEMENTS*)

6502 的 X，Y 暫存器可用來機動地處理陣列元素。這種動作稱爲索引。當你使用 X 索引或 Y 索引時，會產生下列動作：

(1)把所要的索引暫存器的值加到指令中的位址去。



" PICK UP 10TH ELEMENT TO GIVE TO APPLE "

(2)這個位址就是眞正所用的位址。要使用 X 索引則文法爲：

＜符號＞＜位址式子＞，X

例如：

```
LDA LBL,X
STA ARRAY,X
ADC $1,X
SBC $FFFF,X
```

而使用 Y 索引，則文法如下：

＜符號＞＜位址式子＞，Y

例子：

```
LDA LBL,Y
STA ARRAY,Y
ADC $1,Y
SBC $FFFF,Y
```

看看下列例子：

```
LDX #$1
LDA ARRAY,X    ;LOADS ACC FROM LOCATION ARRAY+$1
LDY #$FF
STA STRING,Y   ;STORES ACC AT LOCATION STRING+$FF
```

這程式把位置 ARRAY+$1 的值載入累積器中，再把它存入位置 STRING+$FF. 上。這並沒有用到索引暫存器。

索引位址法最重要的地方是可在程式控制下改變 X ，Y 暫存器的值。例如，假設要從位置 ARRAY 開始，清除 256 個位元組，則可如下：

```
         LDX #$0        ;INIT FOR 256 BYTES
         TXA            ;SET ACC = 0
LOOP     STA ARRAY,X    ;STORE ZERO INTO MEMORY LOC
         INX            ;MOVE TO NEXT LOCATION
         BNE LOOP       ;DONE YET?
         BRK            ;IF SO, QUIT
ARRAY    DFS $100       ;ARRAY STORAGE BEGINS HERE
         END
```

此例中 X 暫存器和累積器都載入 $0 ，接著把累積器的值存入位置 ARRAY+X. 上，此時 X 暫存器為 0 ，那存入 ARRAY 位置上。接著 X 暫存器加 1（為 1 不是 0 ），所以 BNE 指令就跳回 LOOP 位置上，再把累積器的值存入位置 ARRAY+X. ，現在 X 為 1 ，因此 ARRAY+$1. 位置上也存 0 ，一直重覆此廻路直到 X 暫存器的值為 $FF，此時 X 再加 1 而得 0 ，使廻路停止。

X ，Y 暫存器只佔 8 個位元，所以用索引位址法只能在 256 個位元組範圍之內。對大部分應用而言，已經足夠使用了。假如需要更大的陣列可用下面的位址法。

間接位址法（*INDIRECT ADDRESSING MODE*）

只有 JMP 指令可用間接位址法，這裡要提的是使用 X 間接索引法和使用 Y 間接索引法的方法。

間接位址是所要的位址值所在的位址。似乎很怪，看看下面例子會清楚些：

```
JMP $800        ;JUMPS TO LOCATION $800
JMP ($800)      ;JUMPS TO THE ADDRESS CONTAINED
                ;IN BYTES $800 AND $801
                ;LOCATION $800 CONTAINS THE
                ;LOW ORDER BYTE OF THE ADDRESS
                ;AND $801 CONTAINS THE HIGH
                ;ORDER BYTE OF THE ADDRESS
```

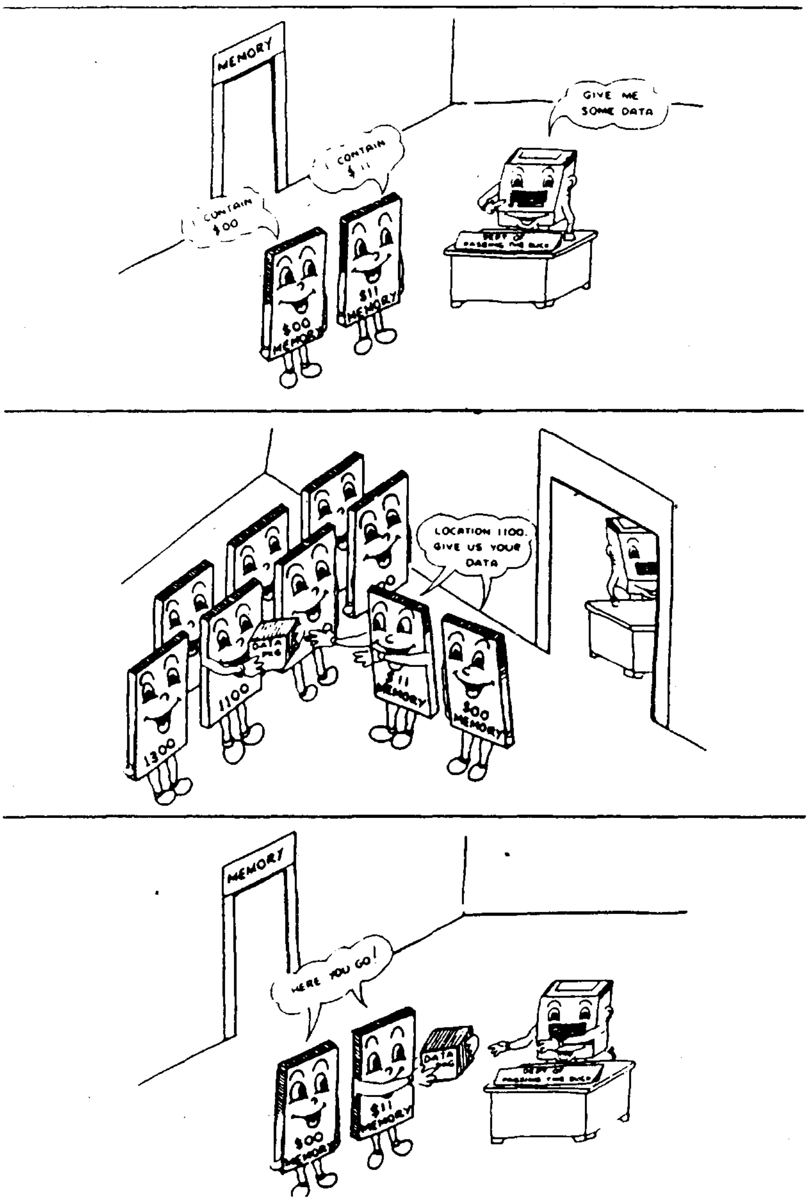
假如位置$800上的值爲$4 而位置$801 上的值爲$09，則JMP（ $800 ）會跳到位置$904。用這種位址法可以來模擬很多語言中都有的 CASE 陳述（ ON...GOTO 和 CASE 是相等的）。下面例子中，假如X暫存器的值爲$0，則JMP會跳到位置$800 ，假如X的值爲$2 ，則跳到位置$900 ，而X若爲$4 ，則跳到位置$1000 。

```
        LDA LOCADR,X
        STA JMPADR
        INX
        LDA LOCADR,X
        STA JMPADR+$1
        JMP (JMPADR)
JMPADR  DFS 2           ;RESERVE TWO BYTES FOR JMPADR
LOCADR  HEX 0008        ;ADDRESS TABLE IN BYTE
        HEX 0009        ;REVERSEE ORDER
        HEX 0010
        END
```

爲什麼要把這麼簡單的問題弄得這麼複雜呢？下列程式也可以作相同的工作啊！

```
        CPX #0
        BNE LBL0
        JMP $800
LBL0    CPX #2
        BNE LBL1
        JMP $900
LBL1    JMP $1000
```

"INDIRECT ADDRESSING"



後面的方法又簡單又短。當然對簡單的例子來說第二種方法可能較好，但記住每多處理一個跳的情形，就要多加 7 個位元組（一個 CPX，一個 BNE 和一個 JMP），而第一種方法卻只要多加兩個位元組即可（一個位址），因此對較複雜的情形，第一種方法可能較好。

除了模擬 CASE 陳述之外，間接跳開亦可用來控制程式的流程。假如要寫一個副程式把 APPLE II 的累積器所存的字元打在螢幕上，在寫好程式之後又希望把程式擴充成可打在印表機等機器上，而且保持程式只有一個入口的方式。我們可以用一個旗號位元組，當旗號爲 0 時，打在螢幕上，當旗號爲 1 時打在印表機上控制。程式如下：

```
PUTCHR  PHA             ;SAVE CHARACTER TO BE OUTPUT
        LDA     FLAG    ;SEE WHERE THE OUTPUT GOES
        BEQ     SCROUT  ;OUTPUT TO THE SCREEN IF 0
        CMP     #1
        BEQ     PRTOUT  ;OUTPUT TO PRINTER IF 1
        CMP     #2
        BEQ     MODEM   ;OUTPUT TO MODEM IF 2
        ETC...
```

假如只有幾種輸出工具則這個方法不錯。但有兩個問題；首先寫在程式時無法預知所有可能會接到計算機的工具，其次是沒有考慮到週邊裝置的起始狀況。對某些週邊裝置而言，第一次使用時必須先跳到起始位址然後再回到正常的入口位址。

用間接位址法就可以解決所有的問題了。不需保留位置做爲旗號而保留兩個位元組來存裝置處理程式（ device handler ）的位址。任意選擇第 0 頁中的 $36 和 $37 二位置來存處理程式的低階和高階位址。假如我們想把字打在螢幕上，就把螢幕的處理程式位址放在位置 $37 和 $38 上，同理

亦可打在印表機上。而眞正的字元輸出程式只有一個指令"
"JMP ($36)" "，這會使程式跳到目前可用的週邊裝置去動作。

第一個問題根本不是問題。這種間接跳到 I/O 處理程式的作法把所有的裝置都考慮進去了。假如現在要把資料打在新的輸出裝置上，只要把新裝置的處理程式的位址放在位置 $36 和 $37 即可。

第二個問題也很簡單。當打開裝置時，它的起始程式的位址就被載入位置 $36 和 $37 中。起始程式設定裝置初值之後再把正常的處理程式位址載入 $36 和 $37 即可。

間接索引位址法
（*INDIRECT INDEXED ADDRESSING*）

只有 JMP 指令可以用間接位址法，其他像載入、存放，比較等指令都不行。這些指令可以用其他兩種混合位址法：由 X 間接索引和由 Y 間接索引法。

間接索引法事實上是兩種方法的混合——間接位法和由 Y 索引法的混合。眞正的位址的求法如下：到運算碼後所指定的位址去取得其值（包括下一個位址的值），然後把 Y 暫存器的值加上取得的值，結果就是眞正的位址。假如 Y 暫存器的值爲 0，就等於用間接位址法。

由 Y 間接索引法有個限制：存位址的位置必須在第 0 頁中。用這種方法的指令佔用兩個位元組：一位元組的運算碼和一位元組的第 0 頁位址；同時如果用符號來代表位址，則必須用 EPZ 來宣告此符號，否則會發生錯誤。

範例：

```
        LDA #$0         ;INIT FOR LOCATION $900
        STA $FE         ;L.O. BYTE IN LOCATION $FE
        LDA #$9         ;
        STA $FF         ;H.O BYTE IN LOCATION $FF
        LDY #$0         ;INIT TO START AT LOCATION $900
        TYA             ;INIT ACC TO ZERO
LOOP    STA ($FE),Y     ;STORE AT LOCATION POINTED AT
        INY             ;BY ($FE,$FF) + CONTENTS OF Y
        BNE LOOP        ;GO TO NEXT, DONE YET?
        BRK             ;IF SO, QUIT
        END
```

這程式就是前面提過的清除記憶體的程式，先前是採用由X和Y索引的方法作的。

到目前爲止仍無法同時處理較256個位元組的位置。以後會提到如何處理記憶體中任何位置的方法。

索引間接位址法

(*INDEXED INDIRECT ADDRESSING MODE*)

在間接位址法中，先找出間接位址再加上Y暫存器的值得到眞正的位置。從名字可以看出是先作間接位址的計算再作索引。

而索引間接法則是先作索引。用X暫存器來索引的文法如下：

<符號>(<位址>, X)

先把位址式子的值加上X暫存器的值，以結果作爲位置，取出該位置和下一位置的值作爲眞正的位址。

當你在第0頁有個指標表，而使用X暫存器的值要處理

記憶體中不同的部分時，由X索引間接就很有用。這個位址法比較不常被使用。其他用法，讀者自行去發掘。

間接位址法是很有用，也是6502 微處理機比其他沒有這種位址法的機器較好的原因，在本書後面會常用到這種位址法，所以一定要了解清楚再看下去。

第九章
邏輯罩幕和位元運算

新指令：

```
AND   ORA   XOR/EOR   BIT
ASL   LSR     ROL     ROR
```

概　論（*GENERAL*）

在眞正計算機應用上，資料並非全被看成字元或數目。另一種常用的資料型態稱爲布林型態。對布林資料型態而言，一般的運算沒有意義，因此需要介紹一些新的指令。

主要有 4 種布林運算：補數，AND，OR 和 XOR 。對 BASIC 程式員來說 AND，OR 應該很熟悉。可是這裡的 AND，OR 與 BASIC 的 AND，OR 有一點點不同。

這裡利用“眞值表”來幫助你了解布林運算。眞值表就是對所有可能的輸入所產生輸出的列表。函數可能只有一個輸出值，可是輸入值卻可以有任意數目。我們的布林運算只允許有一個或兩個輸入值但只產生一個輸出值。因爲布林值不是眞就是假，單一輸入的布林函數只有二種可能，而二個輸入的布林函數則有 4 種可能輸出值（不一定要不同）。

補數函數

(COMPLEMENT FUNCTION)

這個函數只有一個輸入。假如輸入一位元，則送回其補數值。例如輸入爲眞值（ 1 ），則輸出爲假值（ 0 ）；而輸入爲假值（ 0 ），則輸出爲眞值（ 0 ）。函數的眞值表如表 9 － 1 所示。

表 9 － 1　補數函數的眞值表

輸入位元	輸出位元
A	X
0	1
1	0

補數函數有時又稱爲 NOT 函數，又稱爲 1 的補數。

AND 函數（AND FUNCTION）

這函數需要兩個輸入值，而送回一個輸出值。假如二個輸入都是眞則送回眞值，否則送回假值。眞值表如下表 9－2 所示。

表 9－2　AND 函數的眞值表

輸入位元 A	輸入位元 B	輸出位元 X
0	0	0
0	1	0
1	0	0
1	1	1

AND 函數常被用在比較指令中（例如　IF((A=B)　) AND (C<=D))）　AND 函數有罩幕的功能。假如需要可以把某位元變成 0，例如 A 爲變數，假如要使輸出變成 0，則令 B 爲 0 即可，假如 B 爲 1 則輸出值與 A 相同。從表中可看出，若 B 爲 0 則輸出必爲 0，若 B 爲 1，則輸出即等於 A。

OR 函數（*OR FUNCTION*）

這個函數也需要兩個輸入值而送回一個輸出值。假如 A 或 B 有一者（或二者都）爲眞，則送回眞值，否則送回假值。眞值表如下：

表 9－3　OR 函數的眞值表

輸入位元		輸出位元
A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

OR 函數也常用在比較指令中（如 "IF(A = B) OR (C <= D)"），同時 OR 函數亦也有罩幕作用，可以強迫使位元變成 1。假如 A 爲變數，當 B 爲 0 時，A OR B 會使輸出與 A 相同；若 B 爲 1，則 A OR B 會送回 1！

XOR 函數（*EXCLUSIVE-OR FUNCTION*）

同樣也是二個輸入值，一個輸出值的函數。假如 A 和 B 不同時爲眞或假，則送回眞值，否則送回假值，眞值表如下:

表 9－4　XOR 函數的眞值表

輸入位元		輸出位元
A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

XOR 函數有二點特性，第一是可用來取輸入值的補數。假如B是0，則輸出值與A相同，若B為1，則輸出值為A的補數。

XOR 是個"不等於"函數，假如A等於B，則輸出值為假，否則為真。6502 的 CMP 指令也可以作測試相等的動作，可是它會改掉進位旗標的值，而 XOR 不會改掉旗標的值。

位元字串運算（*BIT STRING OPERATIONS*）

到目前為止所討論的輸出和輸入都是一個位元，可是 6502 都是8位元一起運算，因此這些邏輯運算都以8位元來定義。

作法是一位元一位元地作運算。首先拿出一個位元組，從第0位開始，對應另一位元組的第0位作運算，結果放在輸出的第0位，接著是第1位，直到8個位元都作過為止。如下：

```
(10011110) AND (11000111)  =  (10000110)
(11110000) OR  (00001111)  =  (11111111)
(11001100) XOR (11110000)  =  (00111100)
           NOT (11011011)  =  (00100100)
```

邏輯運算的指令
(*INSTRUCTIONS FOR LOGICAL OPERATIONS*)

AND指令 (*AND INSTRUCTION*)

6502 允許你對累積器的值和記憶體中某位置的值作 AND 運算，結果放在累積器中且改變Z和N旗標。指令的符號爲 AND。

範例：

LDA	#$FF	——把值 $FF 載入累積器	11111111
AND	#$OF	——累積器和 $F 取 AND值	00001111
		——結果存在累積器 等於 $F	00001111

LDA	#$2F	——把值 $2F 載入累積器	00101111
AND	#$01	——累積器和 $1 取AND值	00000001
		——結果 $1 存在累積器	00000001

假如位元 7 爲 1 則 N旗標就爲 1，假如結果爲 0 則 Z 旗標就爲 1。 AND 指令有個用途：可用來測試某位元是否爲 1。例如想知道累積器的位元 0 是否爲 1，只要對累積器和 $1 取 AND 值，假如位元 0 爲 1，則結果(在累積器中)一定爲 1，而且 Z 旗標爲 0，你可以用 BNE 指令來測試。如果累積器的位元 0 不爲 1，則結果爲 0，則 BNE 測試就失敗。

除了會改變累積器的值之妥， AND 函數的位元測試特

性非常有用。假如只是要測試位元可用另一 BIT 指令，又方便又不會改掉累積器的值。 BIT 指令對累積器和某一絕對位置或第 0 頁上的位置作 AND 動作，但結果不放在累積器中，而是用來設定 N, Z 和 V 旗標。規則如下：

(1)位置上值的位元 7 載入 N 旗標。（不是結果的位元 7）。

(2)位置上值的位元 6 載入 V 旗標。

(3)視結果來設定 Z 旗標（與 AND 指令相同）。

在輸出入的控制和交談（ hand-shaking）時， BIT 指令特別有用。以後再詳細討論 BIT 指令。

ORA指令 （*ORA INSTRUCTION*）

6502 的 OR 指令為 ORA。

範例：

LDA	#$00	── 值 $0 載入累積器中	00000000
ORA	#$FF	── 對累積器和 $FF 取 OR	11111111
		值，結果 $FF 放在累積器中	11111111

LDA	#$04	── 值 $4 載入累積器中	00000100
ORA	#$30	── 對累積器和 $30 取 OR	00110000
		值，結果為 $34 放在累積器中	00110100

XOR/EOR指令 (*XOR/EOR INSTRUCTION*)

標準的 6502 XOR 指令符號是"EOR"，可是因爲"XOR"很通用，所以 HSA 提供 2 種表示法"XOR"和"EOR"，都是代表 XOR 函數，意思相同，隨便你用。

範例：

```
LDA  #$AA——值 $AA 載入累積器           10101010
XOR  #$01 ——對累積器 $01 取 XOR         00000001
          ——值，結果爲 $AB 放在         --------
          ——累積器中                    10101011

LDA  #$AA——值 $AA 載入累積器           10101010
EOR  #$01 ——對累積器和 $01 取           00000001
          ——XOR 值，結果爲 $AB          --------
          ——存在累積器中                10101011
```

取累積器的補數
(*COMPLEMENTING THE ACCUMULATOR*)

6502 並沒有基本的補數指令，可是可用 XOR 指令來作。對累積器和 $FF 取 XOR 值，就可得累積器值的補數。

範例：

```
LDA  #$00 ——值 $00 載入累積器中       00000000
XOR  #$FF——取 $FF和累積器的 XOR      11111111
              值，結果為 $FF ，存      --------
              在累積器中              11111111

LDA  #$AA——值 $AA 載入累積器          10101010
XOR  #$FF——取 $FF 和累積器的          11111111
              XOR值，結果為 $55        --------
              ，存在累積器中          01010101

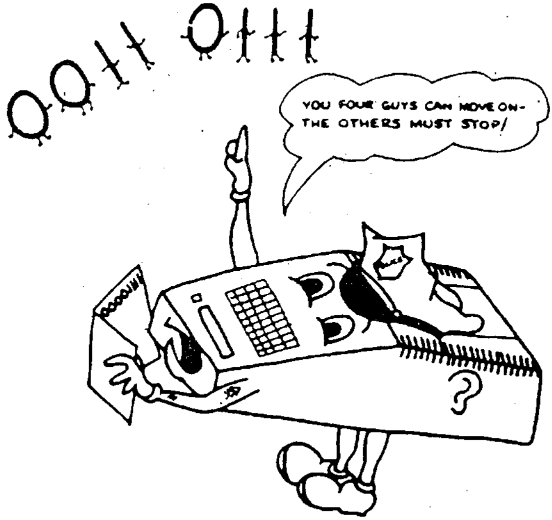
LDA  #$55 ——值 $55 載入累積器的        01010101
XOR  #$FF——取 $FF 和累積器的          11111111
              XOR值，結果為 $FF        --------
              存在累積器中            10101010
```

罩幕運算（*MASKING OPERATIONS*）

上面所提的指令究竟用在那裡？又如何使用？前面提過可用來設定特定位元（用 ORA 指令），清除特定位元（用 AND 指令）和改變特定位元（用 XOR/EOR ），可是它們的價值究竟在那裡？

罩除（*MASKING OUT*）

假設位置 VAR 上有兩種不同值，一在低階尼波（ nibble ），一在高階尼波。在某種情況，我們只對低階的 4 位元有



"MASKING OUT"

興趣，而要把高階的4位元變成0。先把 VAR 的值載入累積器，再對累積器和值 $0F 作 AND運算，則高階 (H.O.) 尼波就被罩除變成0，而低階 (L.O.) 尼波則不變。

範例：

```
LDA  VAR  ——把 VAR 的值載入累積器
LND  #$0F——把高階尼波罩除，留下低階尼波
```

另一個常碰到的問題是聚集資料（ packed data ）。假設爲節省空間，把8個布林值聚集成一位元組。現在想測試某一旗標是否被設定，把此位元組載入累積器，再用 BTR 和 BFL 指令並不夠，因爲BTR（和BNE相同）當任何一位元爲1時就會發生動作，而 BFL 則只有當所有位元都是0

時才會發生動作。我們需要的是能測試某一位元的方法。可以用 AND 指令來做。要測試某一位元，只要把其他位元都罩除，假如被測試的位元爲0，則Z旗標會被設定，而可用 BFL 來測定這情形，假如被測試的位元爲1，則結果會有一位元爲1，而可用 BTR 來測試這情形。

範例：

測試位元＃0

```
LDA BITS
AND #%1
BTR THERE
```

測試位元＃1

```
LDA BITS
AND #%10
BTR THERE
```

測試位元＃2

```
LDA BITS
AND #%100
BTR THERE
```

測試位元＃3

```
LDA BITS
AND #%1000
BTR THERE
```

測試位元＃4

```
LDA BITS
AND #%10000
BTR THERE
```

測式位元＃5

```
LDA BITS
AND #%100000
BTR THERE
```

測試位元# 6

```
LDA BITS
AND #%1000000
BTR THERE
```

測試位元# 7

```
LDA BITS
AND #%10000000
BTR THERE
```

AND指令的另一個用法是在MOD函數（MOD 是餘數函數 X MOD Y則送回X除以Y的餘數）。累積器與$1作AND'，則得到除以2的餘數，和$3作AND' 則得除以4的餘數，與$7作AND'，則得除以8的餘數，與$F作AND'，則得除以16的餘數，與$1F作AND'，則得除以32的餘數，與$3F作AND'，得除以64的餘數，與$7F作AND'，得除以128的餘數，與$FF作AND'，則爲原來的值。

這特性可用在許多地方。如前面測試位元的例子，程式員必須知道要測試那個位元。可以把測試位元的步驟寫成副程式，而以參數來指明要測試那個位元，並且建立有多種不同值的表，當參數傳送到副程式時，就放在X暫存器，而用X索引位址法來決定要用表中那一個值作爲罩子（mask.）。

範例：

```
TSTBIT  LDA BITS
        AND TBL,X
        RTS


TBL     BYT %00000001
        BYT %00000010
        BYT %00000100
        BYT %00001000
        BYT %00010000
        BYT %00100000
        BYT %01000000
        BYT %10000000
```

現在要測試某一位元，則把要測試位元號碼（0－7）載入X暫存器，然後 JSR TSTBIT. 。假定Z旗標被設定，則表被測試位元為0 否則為1 。假如X暫存器的值不在0～7之間時會如何呢？此時記憶體中其他位置的值會被當作罩子，結果是沒有用的！因此需要確定X暫存器的值在0～7之間。有兩種作法，第一種是先把X暫存器與8作比較，如果大於等於8則程式停止執行。第二種方法是把X暫存器和$7作 AND，結果會是X暫存器值除以8的餘數。假如當X暫存器的值為8，你要讓它測試位元0，則第二種方法可用並且較清楚。可看出來， AND' 指令在增加，減少和加法時可用來強迫循環！（'wrap around'）

X暫存器不能直接與記憶體位置作 AND ，所以必須先把X暫存器的值移到累積器，再與表中某值作 AND'，而把結果再移回X暫存器。其實可以把參數直接傳到累積器就可以了！如下面：

```
TSTBIT  AND #%0111
        TAX
        LDA BITS
        AND TBL,X
        RTS

TBL     BYT %00000001
        BYT %00000010
        BYT %00000100
        BYT %00001000
        BYT %00010000
        BYT %00100000
        BYT %01000000
        BYT %10000000
```

AND 指令也可用來某一布林值"假"值。下列程式是把位元組 'BITS.' 中某一位元變成0，與前面相同，累積器中指明要改變那一位元。

範例：

```
SETFLS  AND #$7
        TAX
        LDA BITS
        AND TBLS,X
        STA BITS
        RTS

TBLS    BYT %11111110
        BYT %11111101
        BYT %11111011
        BYT %11110111
        BYT %11101111
        BYT %11011111
        BYT %10111111
        BYT %01111111
```

注意這程式與上一個程式有二點不同。首先在作完 AND 運算之後值存回 BITS 中，如此待會還可以用。其次是表中的資料恰好相反，因為我們只是把某位元變成 0 而其他位元並不希望被罩除。

罩進（*MASKING IN*）

6502 的 ORA 指令可用來設定位元為 1 。程式如下：

```
SETRUE  AND #$7
        TAX
        LDA BITS
        ORA TBL,X
        STA BITS
        RTS

TBL     BYT %00000001
        BYT %00000010
        BYT %00000100
        BYT %00001000
        BYT %00010000
        BYT %00100000
        BYT %01000000
        BYT %10000000
```

同理索引傳送到累積器中且結果放回 BITS. 中。

除此之外， ORA 還有其他用途。例如可用來測試記憶體中二個或多個位元組是否都爲 0 。只要把第一個位元組載入累積器中，然後與接下來的位元組作 OR 函數，如果最後結果 Z 旗標爲 1 則表示所有測試的位元組都是 0 ，假如有某一位元組不爲 0 ，則 Z 旗標爲 0 。

範例：

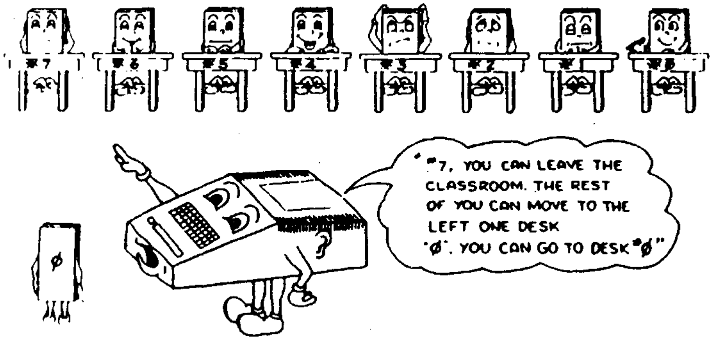
```
LDA BYTE1
ORA BYTE2
ORA BYTE3
ORA BYTE4
 .    .
 .    .
 .    .
ORA BYTEn
BEQ ALLZER
```

移動和旋轉指令
（*SHIFT AND ROTATE INSTRUCTIONS.*）

6502 有 4 個移動和旋轉指令，分別是：算術往左移，邏輯往左旋轉和往右旋轉。這些指令的最簡單形式是在累積器上運算。是 6502 累積器位址法。

算術往左移 (ASL) 指令
（*ARITHMETIC SHIFT LEFT (ASL) INSTRUCTION*）

這指令把累積器所有的位元都往左移一位，位元 0 移到位元 1 ，位元 1 移到位元 2 等，將" 0 "移到位元 0 ，而位元 7 移到進位旗標。它的符號是' ASL '



"ASL INSTRUCTION"

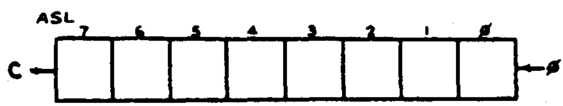
當要移動累積器的值，則指令不需要有運算元。

範例：

把低階尼波移到高階尼波

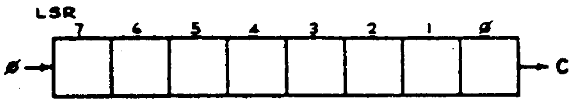
```
LDA  VALUE
ASL
ASL                 ;FOUR SHIFTS MOVE THE L.O.
ASL                 ;FOUR BITS INTO THE H.O.
ASL                 ;FOUR BITS (L.O. FOUR BITS
STA  VALUE          ;BECOME ZERO)
```

假如需要移動多位元位置，則可執行 ASL 多次。



邏輯往右(LSR)指令
(*LOGICAL SHIFT RIGHT (LSR) INSTRUCTION*)

這指令把資料往右移一位元。"0"被移到位元7，位元7移到位元6……等，而位元0移到進位旗標。假如你有二個 BCD 數字而想分成二位元組(即低階尼波到第一位元組而高階尼波到第二個位元組)，把低階尼波弄到第一個位元組很簡單，只要把值載入累積器中，而與 $F 作 AND，再把結果存回第一個位元組即可。



範例：

```
LDA VALUE
AND #$F
STA LOC1
```

而把值和 $F0 作 AND' 並不能完全得到高階尼波，因為我們所要的值仍在累積器的高階尼波中。必須用 LSR 指令把資料往右移動4位，到低階尼波，然後再存回第二個位元組。

範例：

```
LDA VALUE
AND #$F0
LSR
LSR
LSR
LSR
STA LOC1+$1
```

因爲自動把 0 移到位元 7，所以作了四次 LSR 指令後，累積器的高階尼波都變成 0，就無需再和 #$F 作 AND 了!

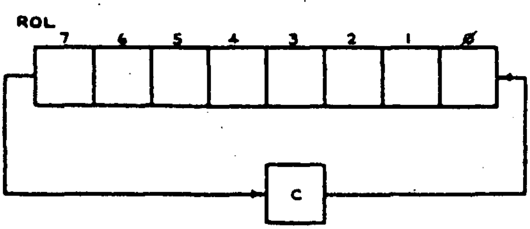
```
LDA VALUE
LSR
LSR
LSR
LSR
STA LOC1+$1
```

最後的程式如下：

```
LDA VALUE
AND #$F
STA LOC1
LDA VALUE
LSR
LSR
LSR
LSR
STA LOC1+$1
```

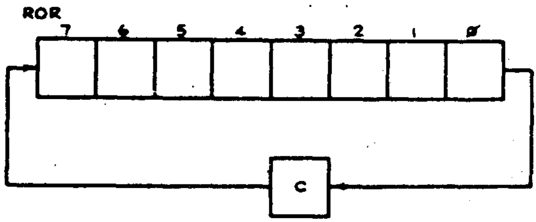
往左旋轉(ROL)指令：
(*ROTATE LEFT (ROL) INSTRUCTION*)

ROL 與 ASL 很類似，除了一點：ASL 是移 0 到位元 0，而 ROL 是把進位旗標移到位元 0。因此 ROL 是把位元 0 移到位元 1 移到位元 2，……，位元 7 則移到進位旗標，而進位旗標移到位元 0。如果連續作九次 ROL 則會再得到原來的值指令符號是 ROL 。



往右旋轉(ROR)指令
(ROTATE RIGHT (ROR) INSTRUCTION)

把累積器往右旋轉而把進位旗標移到位元 7，位元 0 移到進位旗標，符號爲ROR。與 ROL 一樣連續作九次 ROR. 會得到原來的值



移動和旋轉記憶體位置
(SHIFTING AND ROTATING MEMORY LOCATIONS

到目前爲止的旋轉和移動指令都是對累積器作的。6502 也有對記憶體位置作移動和旋轉的指令。假如指令的運算元欄不是空白，則代表一個絕對位置，位置上的值將被移動或旋轉，情形和累積器相同。同時也可以使用 X 索引位址法。

範例：

```
ASL LOC1    -SHIFTS MEMORY LOCATION LOC1 LEFT
LSR TEMP    -SHIFTS MEMORY LOCATION TEMP RIGHT
ROL LBL+$1 -ROTATES MEM LOC. LBL LEFT
ROR X+$1    -ROTATES MEM LOC. X+$1 RIGHT
ASL         -SHIFTS THE ACCUMULATOR LEFT
LSR         -SHIFTS THE ACCUMULATOR RIGHT
ROL         -ROTATES ACC TO THE LEFT
ROR         -ROTATES ACC TO THE RIGHT
```

用ASL來作乘法
（*USING ASL TO PERFORM MULTIPLICATION*）

把數字往左移一位即等於乘以其底數。例如把十進位數 93，往左移一位成 930，就等於乘以底數（ 10 ）。同理二進數往左移一位就等於乘以 2，移二位等於乘以 4，移三位等于乘以 8 等等。

範例：

(1)累積器的值乘 8

```
ASL  ; 乘 2
ASL  ; 乘 4
ASL  ; 乘 8
```

(2)累積器的值乘 32

```
ASL  ; 乘 2
ASL  ; 乘 4
ASL  ; 乘 8
ASL  ; 乘 16
ASL  ; 乘 32
```

同時可用 BCS 指令來測試溢位，當進位旗標爲 1 時表示溢位。如下例一般：

範例：

累積器乘以 16，同時測試溢位

```
ASL
BCS ERROR
ASL
BCS ERROR
ASL
BCS ERROR
ASL
BCS ERROR
```

有時候要乘以不是 2 的乘冪的數，可以把動作分成九個步驟，再把各步驟的和加起來。例如累積器乘以 3 時，可先作累積器乘以 2 ，再作累積器乘以 1 ，二者再加起來。

範例：

乘以 3

```
STA TEMP ;MAKE A TEMPORARY COPY
ASL      ;MULTIPLY ACC BY TWO
CLC      ;ADD IN THE ORIGINAL VALUE
ADC TEMP ;TO GET 2×ACC + ACC = 3×ACC
```

又如乘以 6 ，可以分成乘以 2 ，及乘以 4 兩個步驟。

範例：

乘以 6

```
ASL      ;GET ACC×2
STA TEMP ;AND SAVE
ASL      ;MULTIPLY ACC BY FOUR
CLC      ;ADD IN TEMP VALUE
ADC TEMP ;TO GET 2×ACC + 4×ACC = 6×ACC
```

另一個常用的乘法是乘以 10 ，可分成乘以 8 和乘以 2。

範例：

乘以 10

```
ASL       ;MULTIPLY BY TWO
STA TEMP  ;SAVE
ASL       ;MULTIPLY BY FOUR
ASL       ;MULTIPLY BY EIGHT
CLC       ;ADD IN TEMP VALUE
ADC TEMP  ;TO GET 10×ACC
```

用移動來聚集資料 (*USING SHIFTS TO UNPACK DATA*)

第 2 章曾提到聚集的資料。二個 BCD 數字可聚集成一位元組，8 個布林值也可以集成一位元組。聚集可以節省位置，但較複雜且執行速度較慢。

可以使用把位元組中不要的位元罩除的方式打散 (unpacked) 資料，再往右移到右邊位置。例如 BCD 數字，有二欄：高階十進位數和低階十進位數。假如你要的是低階十進位數，只要把數值和 $F 作 AND 即可，這麼一來不要的位元都已被罩除且已向右看齊。如果是要高階十進位數就複雜一點，作完 AND 之後，要再把資料往右移 4 位，而左邊 4 位元自動補 0 。

不僅在 BCD 中要作資料聚集，有時候可能在一位元組中會有三欄資料，例如位元 7 是布林值，位元 4 ， 5 ， 6 是 AApple slotot 號碼而在低階尼波為一 16 進位數。要拿出左邊 4 位元較簡單，只要與 $1F AND 即可，而拿出中間三位元則較複雜，首先把累積器往右移 4 位，其次與 $7 與 AND' 即可。

範例：

取出中間三位元

```
 A VALUE
 R .           ;SHIFT RIGHT FOUR TIMES
 R             ;TO RIGHT JUSTIFY FIELD
 SR            ;AND ELIMINATE L.O.
 SR            ;NIBBLE
AND #%0111     ;MASK OUT BOOLEAN VALUE
```

要取出那個布林值只要作 7 次 LSR 即可。另外有一種更好的方法，先將累積器與 $80 作 AND ，再往左移一位，將布林值移到進位旗標，最後作往左旋轉，將布林值從進位移到位元 0 即可。

範例：

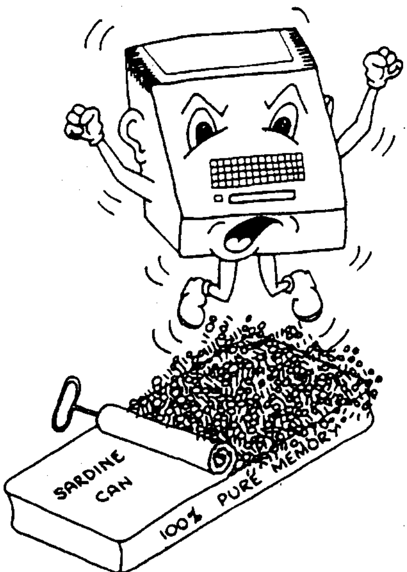
取出布林值

```
LDA VALUE
AND #$80
ASL
ROL
```

假如你只是要測試布林值，就不需作向右看齊的動作可直接用 BTR/BFL 指令測試即可（在 AND 之後），或在 LDA 之後用 BMI/BPL 測試亦可。

用旋轉和移動來聚集資料
(*USING SHIFTS AND ROTATES TO PACK DATA*)

使用前面打散資料的方法還不夠，也要能聚集資料才可以。這比較困難，首先把資料要存的位置先變成 0，（可用 AND 指令來作），其次把要放入的資料先移到正確的位置，最後用 OR' 指令把資料移入變成 0 的欄位！



"PACKING DATA"

範例：

前例中聚集 SLOT 號碼

```
PHA                   ;SAVE DATA TO BE PACKED
LDA VALUE
AND #%10001111        ;MASK OUT SLOT # FIELD
STA VALUE             ;SAVE
PLA                   ;RESTORE ACC
ASL                   ;ALIGN FIELDS
ASL
ASL
ASL
ORA VALUE             ;PUT INTO VALUE
STA VALUE
```

以後會再討論聚集的技巧！

Retro Workshop
果粉工作室

第十章
多重精確度運算

概 論 (*GENERAL*)

到目前爲止，所有的運算都是只對8位元而作。有時候卻需要多於8位元的運算。6502 限制每次只能處理8位元，如果要處理16或32位元，則需分成多個8位元運算。例如，16位元加法，可用兩次8位元加法來處理。

多重精確度的邏輯運算
(*MULTIPLE-PRECISION LOGICAL OPERATIONS*)

多重精確度的邏輯運算(AND, OR, 和 XOR) 最容易處理。假設有二個16位元的運算元分別在位址A, A1和B, B1上。A及B的邏輯運算就等於(A AND B)，(A1 AND B1) 。即把A和B作AND，放在結果的低階位元組，再把A1和B1作AND，放在結果的高階位元組。

範例：A和B作AND，結果存在C

```
LDA A
AND B
STA C
LDA A+$1
AND B+$1
STA C+$1
```

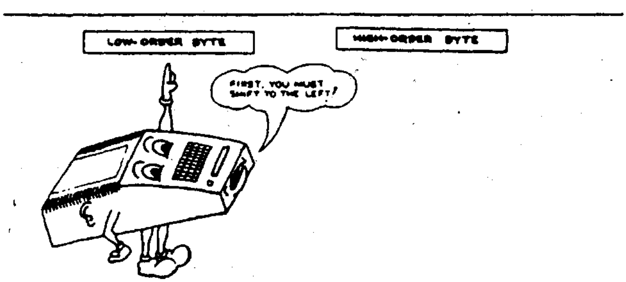
ORA 和EOR 指令的作法類似。

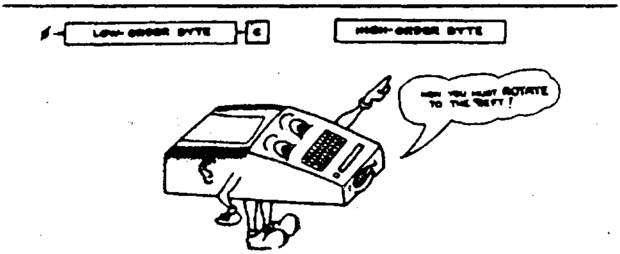
範例：

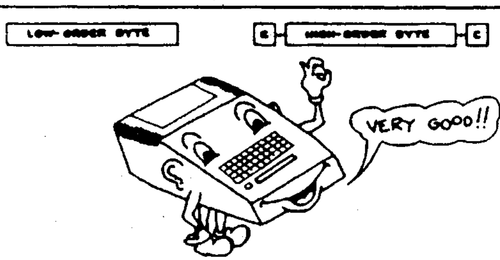
```
LDA A
ORA B
STA C
LDA A+$1
ORA B+$1
STA C+$1
```

```
LDA A
XOR B
STA C
LDA A+$1
XOR B+$1
STA C+$1
```

"TWO BYTE ASL"







多重精確度的移動和旋轉
（*MULTIPLE-PRECISION SHIFTS AND ROTATES*）

移動和旋轉擴充的方式和邏輯運算不太類似。例如 ASL 指令，假如把低階位元組往左移一位，則位元 0 會變成 0，而位元 7 移到進位旗標，再對高階位元組作 ASL ，則從前面位元 7 移過來的進位並沒有如 16 位元 ASL 指令應該作到的移到位元 0 中，而是“ 0 ”被移入位元 0 中。因此應該對高階位元組作 ROL 指令：

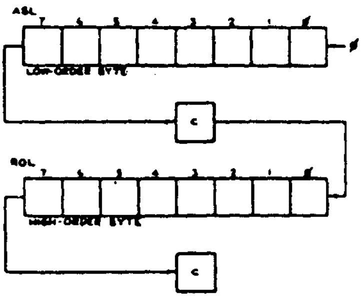
```
ASL LOBYTE
ROL HOBYTE
```

如此從低階位元組移過來的進位旗標，在作 ROL 時就會移到高階位元組的位元 0（正是我們希望的）。三位元組的 ASL ，同理亦可如下處理：

```
ASL BYTE
ROL BYTE+$1
ROL BYTE+$2
```

同理 n 位元組 ASL 可由在上列指令系列中再多加幾個 ROL 指令來達成。

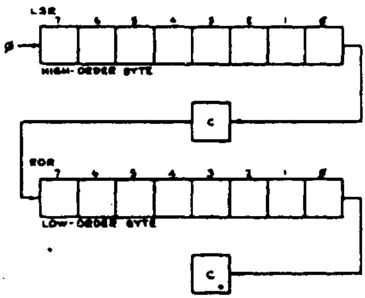
多重精確度的移動和旋轉
二位元組的ASL



多重精確度的邏輯往右移動系列
(*MULTIPLE-PRECISION LOGICAL SHIFT-RIGHT SEQUENCES*)

同上面一樣處理，只是這指令要從高階位元組開始就是了。記住 LSR 指令中，把 0 移入高階位元，而最低階位元移入進位旗標中。二位元組 LSR 的碼如下：

二位元組的LSR



```
LSR BYTE+$1
ROR BYTE
```

同理三位元組 LSR 則如下：

```
LSR BYTE+$2
ROR BYTE+$1
ROR BYTE
```

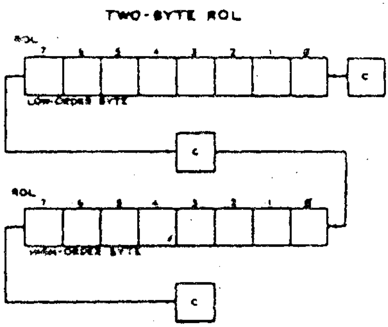
N位元組 LSR 可由對高階位元組作 LSR ，而其餘位元組作 ROR 來合成。

多重精確度的往左旋轉 （*MULTIPLE-PRECISION ROTATE-LEFT SEQUENCES*）

處理的步驟如下：首先把低階位元組旋轉，再旋轉高階位元組。例如16位元的 ROL 程式如下：

```
ROL BYTE
ROL BYTE+$1
```

二位元組的ROL



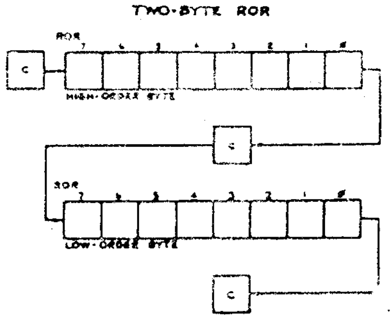
進位旗標移到位元 7 ，位元 7 則移到進位旗標，然後第二個 ROL 指令又把它移到位元 8 。最後位元 15 移到進位旗標，完成了 16 位元的 ROL 動作。而三位元組的 ROL 指令則如下：

```
ROL BYTE
ROL BYTE+1
ROL BYTE+2
```

多重精確度的往右旋轉 （*MULTIPLE-PRECISION ROTATE-RIGHT SEQUENCES*）

跟 LSR 指令一樣，多重精確的 ROR 必須從高階位元組開始旋轉，接著才作低階位元組。16 位元的 ROR 如下：

```
ROR BYTE+$1
ROR BYTE
```



而 3 位元組的 ROR 則如下：

```
ROR BYTE+$2
ROR BYTE+$1
ROR BYTE
```

多重精確度的不帶符號算術（*ARITHMETIC MULTIPLE-PRECISION UNSIGNED*）

只能作一位元組的運算並不夠，常常需要表示大於 255 的值。假如有 16 位元的算術，就可表示從 0 到 16777215之間的值。若是用 4 位元組（32位元），則可表示超過 4 百萬的值。多重精確度的算術的處理方式和多重精確度的邏輯運算的處理方式相同，每次都作一位元組的運算。

最大的問題是要如何抓住當發生溢位（或不足位）時所失去的資料。首先要知道當發生溢位時需要保留住那些資料。8 位元加法的最大值是 $FF + $FF = $1FE（或 510），因此要多使用一個位元來作此擴充的算術運算。

當發生溢位時，進位旗標會等於 1，否則爲 0。因此可用進位旗標作爲第 9 個位元。前面提過，在作加法之前要先清除進位旗標，因爲加法時，會把累積器，運算元和進位旗

標都加在一起，所以要先清除進位，否則會多 1 。

而現在進位旗標成爲 8 位元加法的進位，正好應該加到高階位元組去以得到正確的值。事實上 6502 是這麼作的。例如 16 位元的 OP1 ＋ OP2 ，並把結果存到 RESULT ，其程式如下：

```
CLC              ;ALWAYS BEFORE AN ADDITION
LDA OP1
ADC OP2
STA RESULT

LDA OP1+$1
ADC OP2+$1
STA RESULT+$1
```

注意，現在在各加法之間不要清除進位，因爲進位旗標的值是有用的。三位元組的加法則如下：

```
CLC
LDA OP1
ADC OP2
STA RESULT
LDA OP1+$1
ADC OP2+$1
STA RESULT+$1
LDA OP1+$2
ADC OP2+$2
STA RESULT+$2
```

n 位元組的加法同理可推。

N位元組不帶符號的加法規則

(*RULES FOR UNSIGNED N-BYTE ADDITION*)

(1)不要和其他加法的規則弄混。

(2)在作加法之前要清除進位旗標。

(3)先加第一個位元組，把結果存起來。

(4)再加，第二，第三……個位元組，把結果也存起來，在作這些加法前不要清除進位旗標。

(5)在作完第 n 次加法之後，若是進位旗標爲 1 ，則表示發生溢位，否則進位＝ 0 。

多重精確度的無號減法 （*MULTIPLE-PRECISION UNSIGNED SUBTRACTION*）

減法的作法和加法很像。在作單精確度的減法之前，要先設定進位旗標，同樣地作多重精確度的減法前，也要先設定進位，然後先作低階位元組的相減，接著作高階位元組的相減（結果都要存起來），最後，若是進位＝0，則表示發生不足位，否則進位＝1則表示一切正常。

二位元組的減法範例：
(EXAMPLE OF TWO-BYTE SUBTRACTION:)

```
SEC                 ;ALWAYS!
LDA OPRND1          ;GET L.O. BYTE OF OPERAND #1
SBC OPRND2          ;SUBTRACT L.O. BYTE OF OPERAND #2
STA RESULT          ;SAVE IN L.O. BYTE OF RESULT
LDA OPRND1+$1       ;GET H.O. BYTE OF OPERAND #1
SBC OPRND2+$1       ;SUBTRACT H.O. BYTE OF OPERAND #2
STA RESULT+$1       ;SAVE IN H.O. BYTE OF RESULT
BCC ERROR           ;TEST FOR OVERFLOW
```

要作n位元組的減法，只要多加幾個 SBC 指令即可。記住在一串減法指令中不要清除進位旗標。

多重精確度減法的規則
(*RULES FOR UNSIGNED MULTIPLE-PRECISION SUBTRACTION*)

(1)不要和其它運算的規則弄混。

(2)在作減法之前先令進位＝1。

(3)將低階位元組相減，結果存起來。

(4)再將第二，三……個位元組相減結果也存起來。

(5)作完第n個位元組的相減之後，若C＝0則表示發生不足位，否則一切正常。

多重精確度的有號算術
(MULTIPLE-PRECISION SIGNED ARITHMETIC)

多重精確度的有號算術的作法和多重精確度的無號算術一樣，加法時，先清除進位，然後一個一個位元組相加，而減法時先設定進位，然後一個一個位元組相減。

唯一的不同點是溢位和不足位的測定。與單精確度的帶符號算術一樣，要用V旗標而不是C旗標來測試溢位或不足位。只有當發生溢位或不足位時，V旗標才為1，否則為0。注意在減法中如果發生不足位則，V旗標=1，恰好與不帶符號算術相反（若C=0則表不足位）。

多重精確度的十進位算術
(MULTIPLE-PRECISION DECIMAL ARITHMETIC)

先設定，十進位旗標，就可以作多重精確度的 BCD 算術了。規則和不帶符號的加法，減法都一樣。不要忘記作完之後要清除十進位旗標。6502 並沒有提供帶符號的 BCD算術，不論是單精確度或多重精確度。

多重精確度的增加
(MULTIPLE-PRECISION INCREMENTS)

有時候希望能用 INC 指令來增加二位元組的位置內的值。尤其是和間接Y索引位址法合用時，更是有用。

但 INC 指令並不會影響C旗標，所以無法用C旗標來測試8位元的溢位。 IINC 指令只會改變Z和N旗標的值。

但可以用 Z 旗標當作 C 旗標，因爲 INC 指令只有當前值爲 $FF 時才會發生溢位，而 $FF 加 1 即得 $0，所以當用 INC 指令時，若發生溢位則 Z 旗標會爲 1，因此可用 N 來測試溢位情況。

範例： 16 進位的增加指令

```
        INC LOC
        BNE LBL
        INC LOC+$1
LBL:
```

同理 3 位元組的增加指令如下：

```
        INC LOC
        BNE LBL
        INC LOC+$1
        BNE LBL
        INC LOC+$2
LBL:
```

更多精確度的增加指令處理方式類似。注意，這些增加指令是針對不帶符號的值所作的，帶符號的增加指令也可以合成，但若用 ADC 指令直接加 1 會更簡單。

多重精確度的減指令
（*MULTIPLE-PRECISION DECREMENTS*）

多重精確度的減法和加法一樣有用，也同加法的處理方式類似。有個問題，當從 $0 減 1 成 $FF 時會產生不足位，所以要在作減法指令之前先看 Z 旗標的值，但一定要先把位置的值載入累積器中才有法子測出是否爲 0。16 位元的減的指令如下：

```
        LDA OPRND   ;set Z flag if OPRND is zero
        BNE LBL
        DEC OPRND+$1
LBL     DEC OPRND
```

而三位元組則如下：

```
        LDA OPRND
        BNE LBL1
        LDA OPRND+$1
        BNE LBL2
        DEC OPRND+$2
LBL2    DEC OPRND+$1
LBL1    DEC OPRND
```

若多於三位元組，則 SBC 串列會比 DEC 串列更經濟。同樣地多重精確度 DEC 指令只針對不帶符號的值而作，若要作帶符號值的減少，則用 SBC 串列會更簡單。

多重精確度的不帶符號比較（*MULTIPLE-PRECISION UNSIGNED COMPARISONS*）

多重精確度的比較不像上面的指令有規則可循。每一種比較都有一種全然不同的方法。

測試一16位元值是否爲0。（*TESTING A 16-BIT VALUE FOR ZERO*）

要測試 8 位元的值是否爲 0 ，只要把值載入累積器中然後再測 Z 旗標即可。而測試 16 位元的值，則要先把低階位元組載入累積器中，然後與高階位元組作 OR ，若 16 位元中有某一位元爲 1 ，則 Z 旗標必爲 0 ，否則 Z = 1 。（因此值等于 0 ）

範例：

```
LDA TSTZER
ORA TSTZER+$1
BEQ ISZERO
```

測試一16位元值是否為負數（TESTING A 16-BIT VALUE TO SEE IF IT IS NEGATIVE

這較簡單，只要看位元 15，即高階位元組的位元 7 是否為 1 即可。所以把高階位元組載入累積器中，然後測試 N 旗標，即可知其正負。另外也可用 BIT 指令來測試高階位元組視情況來設定 N 旗標。

相等和不等的測試（TESTING FOR EQUALITY AND INEQUALITY）

相等的測試不太簡單。要分成二部分，首先比較低階位元組，若不等則跳開到程式別的地方，若相等則接著比較高階位元組，若不等則跳到與前面相同的地方，否則二數相等。下列碼當二指定的運算元相等時會跳到 EQUALS，否則跳到 NOTEQL ！

```
        LDA OPRND1
        CMP OPRND2
        BNE NE
        LDA OPRND1+$1
        CMP OPRND2+$1
        BNE NE
        JMP EQUALS
NE      JMP NOTEQL
```

這些碼可用來測試相等或不等。若把 JMP NOTEQL 指令拿掉，則變成 16 位元的 BEQ 指令，當二運算元不等時

會到位置NE 去執行。同樣地可用下列碼來測試 NOT EQUALS

```
       LDA OPRND1
       CMP OPRND2
       BNE NE
       LDA OPRND1+$1
       CMP OPRND2+$1
       BEQ EQL
NE     JMP NOTEQL
EQL:
```

三位或更多位元組的比較同理可推

範例：三位元組的不相等測試

```
       LDA OPRND1
       CMP OPRND2
       BNE NE
       LDA OPRND1+$1
       CMP OPRND2+$1
       BNE NE
       LDA OPRND1+$2
       CMP OPRND2+$2
       BEQ EQL
NE     JMP NOTEQL
EQL:
```

不相等（ <, <=, >, & >= ）較容易處理。前面提過 CMP 指令只是作減法動作而已，唯一不同點是不把結果存下來。多重精確度的 CMP 可以由 CMP 和SBC 指令來模擬。先用 CMP 來比較低階位元組，然後再用 SBC 指令來比較其他位元組，等到都比較其後，就可用 BGE（或 BCS）和 BLT （或 BCC）指令來測試結果了。

範例：

```
X>=Y
-----
LDA X
CMP Y
LDA X+1
SBC Y+1
BGE GE
```

```
X < Y
-----
LDA X
CMP Y
LDA X+1
SBC Y+1
BLT LT
```

要測試＞或＜＝，則可用前面說過的方法（在有關單一位元組比較的那一章）。唯一的問題是要作二個 16 位元值的測試需要使用太多的碼。另一種較好的方法是仍用 8 位元比較，但利用一些數學知識。我們改變比較運算元的順序，首先看測試 X>=Y 的碼如下：

```
X >= Y
------
LDA X
CMP Y
LDA X+$1
SBC Y+$1
BGE THERE
```

因為 X ≥ Y 就等於 Y ≤ X，所以上面的比較也可用來測試 Y ≤ X 的情況。而測試 X<=Y 的碼如下：

```
X <= Y
------
LDA Y
CMP X
LDA Y+$1
SBC X+$1
BGE THERE
```

這裡以 X 作為要比較的值，且用 BGE 指令。測試"＞

”的碼完全一樣，除了一點：要用 BLT 轉移指令來代替 BGE 指令。

多於二位元組的比較可以使用更多的 SBC 指令來模擬。

記住這些比較都只針對不帶符號的值而作。

帶符號的比較 (*SIGNED COMPARISONS*)

首先要測試＝，≠，零或負數時所用的指令與不帶符號的值的方式相同。但“＜＝”，“＜”，“＞＝”的測試則較難。這裡不說明爲什麼，你只要相信若V旗標和C旗標作 XOR 的結果爲1，則表示比較的結果爲＞＝，否則結果爲“＜”。同時 CMP 指令並不會影響V旗標，所以要用 SBC 指令。

```
              X >= Y
              ------
              SEC
              LDA X
              SBC Y
              LDA X+$1
              SBC Y+$1
              BVS LBL1
              BCS LT
        LBL2  JMP GTREQL
        LBL1  BCC LBL2
        LT:
```

```
              X <= Y
              ------
              SEC
              LDA Y
              SBC X
              LDA Y+$1
              SBC X+$1
              BVS LBL1
              BCS GT
        LBL2  JMP LESEQL
        LBL1  BCC LBL2
        GT:
```

```
        X < Y
        -----
        SEC
        LDA X
        SBC Y
        LDA X+$1
        SBC Y+$1
        BVS LBL1
        BCC GE
LBL2    JMP LESS
LBL1    BCS LBL2
GE:

        X <= Y
        ------
        SEC
        LDA Y
        SBC X
        LDA Y+$1
        SBC X+$1
        BVS LBL1
        BCC LT
LBL2    JMP LESEQL
LBL1    BCS LBL2
LT:
```

這些例子並不一定是最好或唯一的解答。經由試驗和調整的過程，或許你會得到你所需要的最佳組合。

第十一章
基本的輸出輸入

概　論（GENERAL）

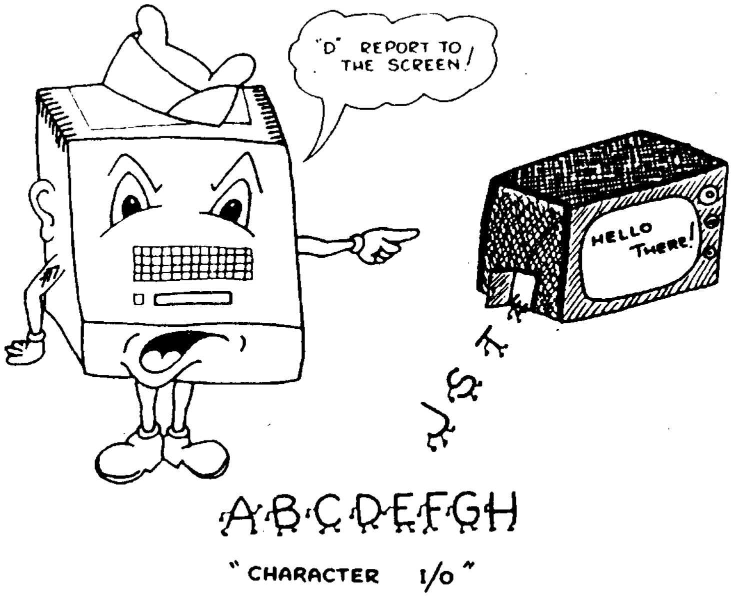
接下去的幾章會舉出一些程式例子並附帶著解釋。若有某些指令不清楚，則你應再復習前面幾章。

字元輸出（CHARACTER OUTPUT）

在 BASIC 中所有輸出都用 PRINT 陳述，但在組合語言中卻沒有 PRINT 陳述。事實上 6502 並沒有提供 I/O 的指令，但要作 I/O 時怎麼辦呢？在 6502 中所有 I/O 儀器都被視爲記憶體位置來處理，所以輸入輸出都用載入和存入指令。 Apple 的銀幕也在記憶體範圍中。

因爲 6502 只能一次處理一位元組，所以輸入輸出也是一字元一字元來作。通常程式會把一字元載入累積器中，然後到輸出字元的副程式去。整數和浮點實數則先轉換成字元再輸出。

APPLE II 的標準輸出儀器是螢幕，就是所謂“與記憶



體相配的螢幕”的一種。每次都以一位元組作為位址來取出要輸出的字元。相反地要放資料在 Apple 的螢幕中，則只要把資料存在螢幕記憶位置上即可。 Apple 監督程式中有在螢幕上輸出字元的副程式。但在螢幕上那個位置呢？就緊接在前一個輸出的字元之後。副程式放在 Apple 監督程式 ROM 中的位置 $FDF0 。用法如下：

```
LDA #"A"
JSR $FDFO
LDA #"B"
JSR $FDFO
LDA #"C"
JSR $FDFO
RTS
END
```

這程式在銀幕上輸出 ABC ，然後跳回監督程式中（或其他的呼叫副程式）。這種輸出非常簡陋！

另一種較好的方法要利用 6502 的索引暫存器。先用 STR 指令把字串先存到某位置，然後令索引暫存器為 1（跳過由假指令 STR 所產生的碼的長度），然後輸出所有字元直到方存器的值大於字元的長度為止。

```
        LDX #$0
LOOP    INX
        LDA STRING,X
        JSR $FDF0
        CPX STRING
        BLT LOOP
        RTS
        :
STRING  STR "I WON! CARE TO PLAY AGAIN?"
        END
```

上例中，X 暫存器的原值為 0，然後增加成 1。在輸出字元之後就要比較 X 暫存器和長度位元組的值，若小於長度位元組，則再輸出另一字元。

有個問題，若字串的長度為 0 怎麼辦？至少會輸出一字元。有時候會有長度為 0 的字串。要能輸出長度為 0 的字串，需使用下列的程式碼：

```
        LDX #$0
LOOP    CPX STRING
        BGE EXIT
        LDA STRING+$1,X
        JSR $FDF0
        INX
        JMP LOOP
EXIT    RTS
STRING  STR "I WON!  CARE TO PLAY AGAIN?"
        END
```

不再增加 X 暫存器的值而改成當載入 A 暫存器時，加個分支到 STRING 的位址上。如此一來當字串的最後字元載入累積器時，X 暫存器的值會等於字串的長度減 1。當 X 暫存器的值等於表示長度的位元組時，副程式就停止執行，然後用 BGE 指令來代替 BEQ 指令。

雖然這方法比前面一個一個字元輸出的方法好，但仍有缺點。若要印出許多行資料時怎麼辦？在 BASIC 中可以再多寫一 PRINT 陳述即可。但在組合語言中，則必須反覆寫上面的串列，這並不太方便。

ASCII 字元集合中沒有包括一特殊字元" ETX "(即 end-of-text)($83, or control-C)，但有時候卻需要輸出這字元，因此可選擇一個幾乎很少輸出到週邊裝置的字元碼代表。('@')' 就合乎這個條件，且在 APPLE II 中的碼爲 $00 ，這可任意選擇這字元碼，但因爲它可用來測試零，所以可用在下面例子：

```
        LDX #$0          ;INIT POINTER TO CHARACTERS
LOOP    LDA STRING,X     ;GET NEXT CHARACTER
        BEQ EXIT         ;IF ZERO, QUIT
        JSR $FDFO        ;OTHERWISE OUTPUT
        INX
        JMP LOOP
EXIT    RTS
STRING  ASC "I WON!  CARE TO PLAY AGAIN?"
        BYT $0
        END
```

注意用 ASC 指令來代替 STR 指令。因爲 STR 假指令先輸出一長度位元組，然後再輸出字串，這並不是我們所希望的。

只要加入一些 carriage returns 在文章中，就可以輸出許多行了。

```
        LDX #$0
LOOP    LDA STRING,X
        BEQ EXIT
        JSR $FDFO
        INX
        JMP LOOP
```

```
;
EXIT     RTS
STRING   ASC "I WON!  CARE TO PLAY AGAIN?"
         BYT $8D
         ASC "(Y/N):"
         BYT $0
         END
```

上例中在結束之前會輸出二行。只要在文章字串中加入CR($8D)即可，就可輸出任意多行。

這程序有個缺點，因爲用X暫存器來接近要輸出的字串元素，因此字串的長度要小於等於255。如果只有一行，則255就夠了，但如果要輸出很多行則255太小事實上，上例中有一個漏洞；如果要輸出多於255個字元，則X暫存器會溢位又變成0而從字串的第一個字元開始輸出，如此一來會形成不停的廻路且印出許多沒有的資料。要避免這錯誤，則用下列的程式碼：

```
         LDX #$0
LOOP     LDA STRING,X
         BEQ EXIT
         JSR $FDF0
         INX
         BNE LOOP
;
EXIT     RTS
STRING   ASC "> 255 CHARACTERS HERE"
         BYT $0
         END
```

在例子中，用 BNE LOOP 指令來代替 JMP LOOP 指令。若X暫存器發生溢位，則會跳出副程式而不再繼續作。這裡的改正並不是允許你輸出多於255個字元，而是當輸出了255個字元之後就會自動停止。因此，字串的某部分仍沒有被輸出，可是你的程式卻不會再有不停的廻路和在螢幕上印出一大堆無用的資料了！

要輸出長度大於255的字串則要用一個 16 位元的指標

。也就是說要用由Y間接索引的位址表示法。下面的副程式可以允許你輸出任意長度的字串：（少於65535個字）

```
          LDA #STRING      ;MOVE ADDRESS OF STRING
          STA $0           ;INTO LOCATIONS $0 AND
          LDA /STRING      ;$1
          STA $1
          LDY # 0          ;INIT Y REGISTER
LOOP      LDA ($0),Y
          BEQ EXIT
          JSR $FDFO
          INY
          BNE LOOP         ;IF NO OVERFLOW, KEEP IT UP
          INC $1           ;INCREMENT BEYOND 8 BITS
          BNE LOOP
EXIT      RTS
STRING    ASC "STRING OF ANY LENGTH"
          HEX 00
          END
```

這個程式有一些特點，首先是增加Y暫存器而非位置$0的值。這會節省一個位元組的碼同時速度也較快。其次在執行副程式之前要先設定位置$0和$1的值。雖然可能需要更多的碼來寫這個副程式，但卻値得，因爲這副程式可變成通用的副程式。看下列程式：

```
PRTSTR    STA $0
          STY $1
          LDY #$0
LOOP      LDA ($0),Y
          BEQ EXIT
          JSR $FDFO
          INY
          BNE LOOP
          INC $1
          BNE LOOP

EXIT      RTS
```

有了這個副程式，你只需把輸出字串的位址載入累積器和Y暫存器（高階位元組在Y暫存器，低階位元組在累積器），然後使用 JSR PRTSTR 指令即可。

範例：

```
        LDA #STRING
        LDY /STRING
        JSR PRTSTR
        RTS
STRING  ASC "STRING OF ANY LENGTH"
        BYT $0
        END
```

現在要輸出一字串只要 3 行指令就可以了。但這方法仍有兩個缺點，首先三行仍比一行多二行，其次這方法需要把資料傳送到累積器和 Y 暫存器，最好是可免除用暫存器來傳送參數。

最後一種方法是利用 6502 堆疊把字串的位址傳送給副程式。看下列程式：

```
JSR PRINT
ASC "HELLO THERE"
HEX 00
RTS
END
```

這程式會跳到副程式 PRINT 中，然後再回到下一個指令——即字元" H "，但這不是所希望的，字串應該放在不會被執行到的地方。前面提過，呼叫副程式時，送回位址 -1 就被壓入堆疊中。當位址被彈出且加 1 後就指到" HELLO "中的" H "。用這個指標就可以輸出所有的資料直到碰到" 00 "爲止。當碰到 $0 時，下一個位元組的指令會再把位址再壓入堆疊中，然後執行正常的 RTS 指令。跳回之後，6502 從 $00 後面的程式繼續執行下去。另一種作法是當碰到 $0 時，把位址加 1 ，然後間接地跳到那個位址。這可模擬 RTS 指令且節省了一些空間。最後，副程式 PRINT 更改

如下：

```
PRINT    STA ASAVE        ;SAVE ACC
         STY YSAVE        ;SAVE Y REG
         PLA
         STA ZPAGE
         PLA
         STA ZPAGE+$1
         JSR INCZ
         LDY #$0
PLOOP    LDA (ZPAGE),Y
         BEQ EXIT
         JSR $FDF0
         JSR INCZ
         JMP PLOOP
;
EXIT     JSR INCZ
         LDA ASAVE
         LDY YSAVE
         JMP (ZPAGE)
;
INCZ     INC ZPAGE
         BNE INCZ0
         INC ZPAGE+$1
INCZ0    RTS
         END
```

當呼叫程式時，字串是緊接在 JSR 指令之後，而以 16 進位的 00 作爲結束。

範例：

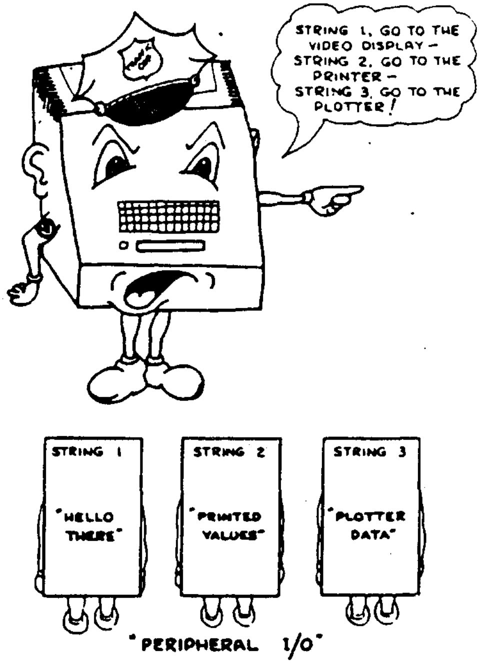
```
JSR PRINT
ASC "I WON!  CARE TO PLAY AGAIN?"
BYT $8D
ASC "(Y/N):"
BYT $0
 .
 .
 .
JSR PRINT
ASC "HELLO THERE, HOW ARE YOU!"
BYT $0
JSR PRINT
BYT $8D
ASC "I AM A SMART COMPUTER!"
BYT $0
 .
 .
ETC....
```

標準輸出和週邊裝置（STANDARD OUTPUT AND PERIPHERAL DEVICES）

到目前爲止都是輸出到銀幕上。要在螢幕上輸出一字母，只要把字元載入累積器中，然後 JSR 到 $FDF0 即可。在組合語言中最好不要用絕對位址，所以可以用 EQU 指令定義一符號性標示，令其值爲 $FDF0 可以用標示 COUT1 代表，因爲在 Apple 監督程式中也是用這個標示，這樣子別人看你的程式就會聯想到螢幕輸出程式了！

有時候要把資料輸出到別種週邊裝置，處理的方式和在螢幕上輸出的方式類似。也是把要輸出的字載入累積器中，然後用 JSR 跳到處理輸出的副程式。副程式的標準位址是 $Cn00 ，其中 n 爲週邊裝置的擴充接點號碼，範圍爲 0 到 7。注意這方法只對板上有 ROM 的"智慧型"週邊裝置有效。若是從電子系統或微型成品買來的"愚笨型"週邊裝置則要用另一套完全不同的"驅動程式軟體"記憶位置"的方法。同時這方法也不適用在 Disk II或 Tape II裝置上，因爲它們用 ROM 來叫出載入程式。假如你在擴充接點 1 上有個印表機界面，則要從印表機上輸出字元，只要把字元載入累積器，然後執行 JSR $C100 即可。

但用 Apple 的標準輸出會更簡單，這時就不用 JSR $Cn00，只要用 JSR $FDED（標示= COUT ）即可。這會使輸出資料送到目前可動（ active ）的週邊裝置。而用命令 PR#n 和 IN#n 就可使某週邊裝置變爲可動的裝置。要模擬 PR#n 命令，首先把擴充接點號碼載入累積器中，然後 JSR 到監督程式中的位置 $FE95 上（爲副程式" OUTPORT "）。同理模擬 IN#n 命令，則把擴充接點號碼載入累積器中



，然後 JSR到位置 $FE8B 上（爲副程式" INPORT "）。若要重新設定 I/O 向量給螢幕或鍵盤（和 PR#0 或 IN#0 作用相同），則把 0 載入累積器，然後跳到所要的副程式。若是模擬 PR#0，則跳到 $FE93，而 IN#0，則跳到 $FE89。

當執行位置 $FDED 上的副程式時，第一個指令爲 JMP ($36)，而位置 $36，$37 的值通常是 $F0 和 $FD，也就是說每當 JSR $FDED（或 JSR COUT）時，就會執行副程式 COUT1。若在輸出字元之前先作 PR#n 命令，則值 $00會被塞入位置 $36，而 $Cn 則被塞入位置 $37上。因此字元就由位置 $Cn00 上的副程式來負責輸出。

```
-SIMULATION OF A PR#3
        LDA #$00
        STA $36
        LDA #$C3
        STA $37

-SIMULATION OF A PR#0
        LDA #$FDFO
        STA $36
        LDA /$FDFO
        STA $37

-CAUSE OUTPUT TO BE ROUTED TO USER ROUTINE
 AT LOCATION $300
        LDA #$300
        STA $36
        LDA /$300
        STA $37
```

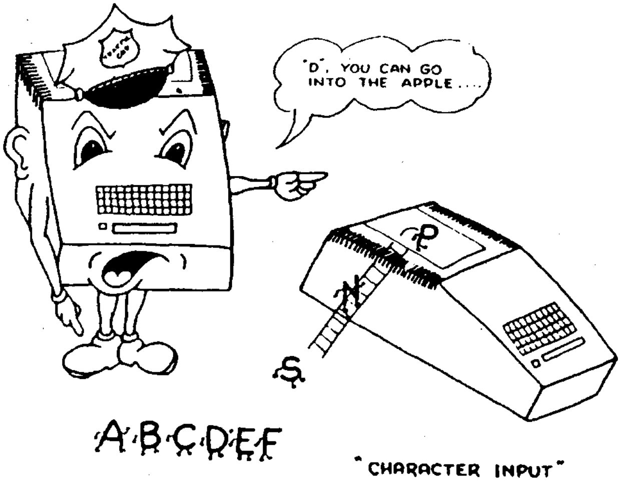
最後的例子很重要，因爲它說明了如何來驅動使用者自行定義的輸出程式，下面即爲此種副程式的例子：

```
        ORG $300
        LDA #DBLVSN
        STA $36
        LDA /DBLVSN
        STA $37
        RTS
:
DBLVSN  JSR $FDFO
        JMP $FDFO
        END
```

字元輸入
(*CHARACTER INPUT*)

與字元輸出一樣，字元輸入也是每次一個字元地處理。Apple II 的鍵盤對使用者的程式而言似乎有二個位置。位置 $C000 中有最近打入的 ASCII 碼，若位元 7 爲 1，則表示打入的鍵爲有效鍵，否則表示尚未打入鍵且位置 $C000 上的資料並無意義。若要再打入其餘的鍵則只要把位置 $C010 的

位元 7 清除爲 0 即可。



因此要從 Apple Ⅱ 鍵盤讀入一個字鍵，要執行下列步驟:

(1)讀位置 $C000 的資料，一直反覆直到位元 7 爲 1 。

(2)把位置 $C000 的資料載入累積器中。

(3)把累積器的值存入位置 $C010 上，使位置 $C000 可再接受下一個輸入。

程式如下：

```
KEYIN LDA $C000
      BPL KEYIN
      STA $C010
      RTS
```

或許你已注意到，以這種方式打入的鍵不會回饋到螢幕上。若要有回饋，則用下列程式

```
TPWRTR JSR KEYIN
       JSR COUT
       JMP TPWRTR
KEYIN  LDA $C000
       BPL KEYIN
       STA $C010
       RTS
COUT   EQU $FDF0
       END
```

若要跳出程式，則按 RESET 鍵即可。

使用 Apple 的鍵盤和螢幕時，監督程式提供了一個非常方便的字元輸入副程式，放在位置 $FD0C 上同時會把目前的游標（ cursor ）位置設定成閃動型式（ flashing mode ）。在打入鍵時，閃動的游標會被在其下面的資料所替代。一種更好的"電動打字機"如下：

```
LOOP   JSR RDKEY
       JSR COUT
       JMP LOOP
RDKEY  EQU $FD0C
COUT   EQU $FDED
       END
```

位置 $FD0C 上的副程式不會把字元回饋到螢幕，所以用 JSR COUT 就可以了。

就如同位置 $FDED 上的副程式透過標準輸出來處理 I/O 參數一樣，位置 $FD0C 的副程式也從"標準輸入"上取得輸入資料。利用 JSR 到位置 $FD0C，就可以從 Disk II ， Mountain Computer's Apple Clock，和外部終端機等週邊裝置讀入資料。

標準輸出和輸入的處理方式有二點不同。首先副程式是從位置 $38和 $39 上取得輸入資料的位置。其次輸入資料會送回累積器中。

要模擬 IN# 命令，先要把所要的擴充接點號碼載入累積器中，然後 JSR 到位置 $FE8B 的副程式。 IN#0 命令可用 JSR $FE89 來模擬。處理輸入時要更小心一些，讀者最好研究一下 Apple ROM 中的監督程式中從位置 $FD0C 到 $FD2E 上的輸入程式。

一行字元的輸入 (*INPUTTING A LINE OF CHARACTERS*)

要輸入一行字元只要連續地讀入一個個字元，然後存到連續的位址上就可以了，直到碰到 carriage return 為止。看起來很簡單，事實上卻有一些問題。例如當按下" backspace " ASCII 碼 $8D。若印出" backspace "，則游標會往後移一格，但事實上你並不希望把 backspace 字元當作文章中的字，而是要刪去前一個打入的字。同時右箭頭鍵(→)和 control-U 效果一樣也不會拷貝游標下的資料，而是送回 ASCII 碼 $95。另外結束功用的 ESC 也沒有作用。所以事實上並不簡單。

幸運的是已經有一個整行輸入的程式可供使用。這程式的位置是 $FD67 稱為" GETLNZ "。呼叫此程式時它會送出一個 carriage return ，同時打出一"提辭"(prompt)字元，接著就可從目前的輸入裝置讀入一行文字。存在位置 $33的字永遠被當作提辭，所以如果你想換新的提辭(例如："："或"－"或"＝")，只要在呼叫 GETLNZ 之前把此字元存入位置 $33 即可。

GETLNZ 有兩個入口點。 GETLN(在位置 $FD6A)在輸出提辭之前不會送出 carriage return 。而 GETLN1 (在位置 $FD6F)則二者都不輸出。這二個入口點都很有用。

當呼叫 GETLNZ , GETLN 或 GETLN1 ，文字在那裡結束呢？所有文字都是從位置 $200 開始順序存放。每行頂多存 256 個字。因此第二頁永遠不能用來存程式碼或資料。從程式 GETLNZ ， GETLN 或 GETLN1 跳回時，X 暫存器

會存有眞正輸入字元的個數（不包括 carriage return ）。 GETLN 程式會回饋所有的輸入，所以使用者可以看到發生了什麼事情，此外也支持 Apple 螢幕的所有特性。而究竟要如何使用這些整行輸入的程式呢？我們下一章再討論。

第十二章
數值的I/O

概　論（*GENERAL*）

有時候也要輸入或輸出數值資料。這章將討論4種數值I/O：

(1) 16進位I/O
(2) 位元組/數值I/O
(3) 整數（16位元或更多）I/O
(4) 帶符號整數（16位元的2的補數）I/O

16進位的輸出（*HEXADECIMAL OUTPUT*）

最簡單的輸出數值資料的型態是16進位數。雖然我們可以自己寫個程式來作這工作，但卻沒有這個需要，Apple監督程式已經提供了一個很好的程式，位置就在$FDDA。這程式把累積器的值印成二個16進位數。累積器的值會被破壞，但不影響其他的暫存器。Apple監督程式中這個程式的名字為PRBYTE，但在使用者程式中卻通常稱為HEX-OUT。應該注意的是16進位輸出和BCD輸出的程式是同

一個程式，因此如果要輸出 BCD 數目，也是用位置$FDDA上的程式。

要輸出大於一位元組的數目（BCD 或16 進位），先把高階位元組載入累積器，然後 JSR HEXOUT ，如此反復地執行直到所有位元組都被處理過了爲止。



"HEX-ING" THE BINARY DATA

爲說明起見，再把監督程式中的 PRBYTE程式列出如下

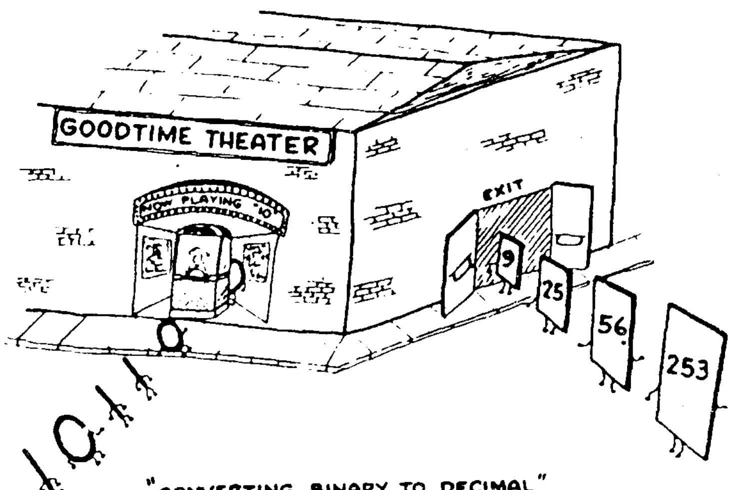
```
PRBYTE  PHA             ;SAVE ACC FOR USE LATER ON
        LSR             ;SHIFT H.O. NIBBLE
        LSR             ;DOWN TO THE L.O. NIBBLE
        LSR             ;CLEARING THE H.O. NIBBLE
        LSR
        JSR PRHEXZ      ;PRINT L.O. NIBBLE AS A DIGIT
        PLA             ;GET ORIGINAL VALUE BACK
PRHEX   AND #$F         ;MASK H.O. NIBBLE
PRHEXZ  ORA #$B0        ;CONVERT TO ASCII
        CMP #$BA        ;IF IT IS A DIGIT FINE, OTHER-
        BLT PRTIT       ;WISE IT MUST BE CONVERTED TO A
        ADC #$6         ;LETTER IN THE RANGE A-F
PRTIT   JMP COUT
COUT    EQU $FDED
        END
```

程式中需要用 CMP #$BA 指令，因爲在 ASCII 字元集合中" A "不是緊接在" 9 "之後。既然 BLT 和 BCC 是相等的，所以當碰到 ADC #$6 時，一定會設定進位旗標。我們把 7 加到累積器中。 $BA 加 $7 得 $C1，即爲 A 的 ASCII 碼！

十進位值的位元組輸出（*OUTPUTTING BYTE DATA AS A DECIMAL VALUE*）

對電腦人員來說，16 進位數目很容易了解，但對其他人則還是使用 10 進位數目較好。監督程式中沒有輸出十進位數（除了 BCD ）的程式，我們必須自己設計。這節中先討論輸出在 0 到 255 之間的一位元組不帶符號整數的方法。

方法很簡單。先把二進位數和 100 比較，若較大或等於 100，則減去 100，直到值小於 100 爲止。每作一次減法之



"CONVERTING BINARY TO DECIMAL"

後，則把某位置的值加1，因此當數目小於100時，其百位上的數字會存在此位置上，然後把值輸出到螢幕。重覆上面步驟，但現在每次減10，且與10作比較，作完之後再輸出到螢幕。現在剩下的值小於10，所以可直接輸出！

除此之外，尚需要一旗標來避免印出左邊的" 0 "。先把某位置的值設爲正數。在輸出數字之前，先檢查此旗標看是否可輸出" 0 "。程式如下：

```
PRTBYT  PHA             ;SAVE REGISTERS
        TXA
        PHA
;
        LDX #$2         ;MAX OF 3 DIGITS (0-255)
        STX LEADO       ;INIT LEADO TO NON-NEG VALUE
PRTB1   LDA #"0"        ;INITIALIZE DIGIT COUNTER
        STA DIGIT
;
PRTB2   SEC
        LDA VALUE       ;GET VALUE TO BE OUTPUT
        SBC TBL10,X     ;COMPARE WITH POWERS OF 10
        BLT PRTB3       ;IF LESS THAN, OUTPUT DIGIT
;
        STA VALUE       ;DECREMENT VALUE
        INC DIGIT       ;INCREMENT DIGIT COUOTER
        JMP PRTB2       ;AND TRY AGAIN
;
PRTB3   LDA DIGIT       ;GET CHARACTER TO OUTPUT
        CPX #$0         ;CHECK TO SEE IF THE LAST DIGIT
        BEQ PRTB5       ;IS BEING OUTPUT
        CMP #"0"        ;TEST FOR LEADING ZEROS
        BEQ PRTB4
        STA LEADO       ;FORCE LEADO NEG IF NON-ZERO
;
PRTB4   BIT LEADO       ;IF ALL LEADING ZEROS, DON'T
        BPL PRTB6       ;OUTPUT THIS ONE
PRTB5   JSR COUT        ;OUTPUT DIGIT
PRTB6   DEX             ;MOVE TO NEXT DIGIT
        BPL PRTB1       ;QUIT IF THREE DIGITS HAVE
        PLA             ;BEEN HANDLED
        TAX
        PLA
        RTS
TBL10   BYT !1
        BYT !10
        BYT !100
;
COUT    EQU $FDED
LEADO   EPZ $0
DIGIT   EPZ LEADO+$1
VALUE   EPZ DIGIT+$1
        END
```

使用這程式時，把要輸出的值載入位置 VALUE 上，然後執行 JSR PRTBYT 。相對於 VALUE 位置上值的十進位數目會輸出到螢幕上。（或其他裝置上）

16位元不帶符號整數的輸出（*OUTPUTTING 16-BIT UNSIGNED INTEGERS*）

用二位元組則可表示在 0 到 65535 之間的不帶符號值。輸出這範圍的值很簡單，只要把前一個程式擴充到可測試 1000 到 10000 範圍的值即可。程式如下：

```
PRTINT  PHA              ;SAVE REGISTERS
        TXA
        PHA
        LDX #$4          ;OUTPUT UP TO 5 DIGITS
        STX LEADO        ;INIT LEADO TO NON-NEG
;
PRTI1   LDA #"0"         ;INIT DIGIT COUNTER
        STA DIGIT
;
PRTI2   SEC              ;BEGIN SUBTRACTION PROCESS
        LDA VALUE
        SBC T10L,X       ;SUBTRACT LOW ORDER BYTE
        PHA              ;AND SAVE
        LDA VALUE+$1     ;GET H.O BYTE
        SBC T10H,X       ;AND SUBTRACT H.O TBL OF 10
        BLT PRTI3        ;IF LESS THAN, BRANCH
;
        STA VALUE+$1     ;IF NOT LESS THAN, SAVE IN
        PLA              ;VALUE
        STA VALUE
        INC DIGIT        ;INCREMENT DIGIT COUNTER
        JMP PRTI2
;
;
PRTI3   PLA              ;FIX THE STACK
        LDA DIGIT        ;GET CHARACTER TO OUUPUT
        CPX #$0          ;LAST DIGIT TO OUTPUT?
        BEQ PRTI5        ;IF SO, OUTPUT REGARDLESS
        CMP #"0"         ;A ZERO?
        BEQ PRTI4        ;IF SO, SEE IF A LEADING ZERO
        STA LEADO        ;FORCE LEADO TO NEG.
;
PRTI4   BIT LEADO        ;SEE IF NON-ZERO VALUES OUTPUT
        BPL PRTI6        ;YET
PRTI5   JSR COUT
PRTI6   DEX              ;THROUGH YET?
        BPL PRTI1
        PLA
        TAX
        PLA
        RTS
```

```
;
T10L      BYT !1
          BYT !10
          BYT !100
          BYT !1000
          BYT !10000
;
T10H      HBY !1
          HBY !10
          HBY !100
          HBY !1000
          HBY !10000
;
COUT      EQU $FDED
LEADO     EPZ $0
DIGIT     EPZ LEADO+$1
VALUE     EPZ DIGIT+$1
          END
```

用這程式時，把要輸出的值載入位置 VALUE 和 VALUE + $1 上，然後 JSR PRTINT ，讓副程式替你完成其餘的工作。這程式相當普通，而且也可擴充到輸出多於二位元組的數目。只要在標示 PRTI2 和 PRTI3 之間多作一次減法來處理最高階位元組就可以了，同時用另一個位元組表設定所要輸出資料的最高位元組的值。最後修改 LDX #$4 成 LDX #$n，其中 n 爲輸出數目的位數減 1 。除此之外，這程式可適用於任何長度的整數。

帶符號16位元整數的輸出 (*OUTPUTTING SIGNED 16-BIT INTEGERS*)

要輸出 2 的補數的帶符號值非常簡單。先檢查它的高階位元，若爲 0 ，則跳到程式 PRINT （上面提過的），若爲 1 則輸出一個 " 一 " ，取值的 2 的補數，然後跳到程式

PRINT。程式如下：

```
PRTSGN  BIT VALUE+$1      ;TEST SIGN BIT
        BPL PRTINT        ;IF POSITIVE, GO TO PRTINT
        PHA               ;SAVE ACC
        LDA #"-"          ;OUTPUT A "-"
        JSR COUT
        SEC               ;TAKE TWO'S COMPLIMENT OF
        LDA #$0           ;VALUE.
        SBC VALUE
        STA VALUE
        LDA #$0
        SBC VALUE+$1
        STA VALUE+$1
        PLA
PRTINT  ——                ;INSERT PRTINT ROUTINE HERE.
```

輸出整數的一種簡單方法
(*AN EASY METHOD OF OUTPUTTING INTEGERS*)

前面提過的十進位輸出程式都很好用，但都需要使用相當多的碼。在 JSR 執行之前，要先把輸出值載入位置 VALUE 和 VALUE + $1 上。所以總共需要 8 到 10 個位元組和四行指令。下面的程式可在 JSR 陳述之後直接指明輸出資料的位置，所以只需多一行和二位元組碼即可，就跟使用 PRTINT I 命令一樣的方便。程式執行的方式如下：

(1)送回位址從堆疊中被彈出，然後存在 VALUE 上。

(2) VALUE 的值加 2，再壓入堆疊中。修改了送回位址以便當跳回時 6502 會回到在 2 位元組的位址之後的入口點。

(3) VALUE 的值減 1，現在它指到在 JSR 指令之後的二位元組位址。

(4)把 (VALUE) 和 (VALUE) + $1 所指的二位元組的值載入 VALUE 中。

(5) VALUE 所指的資料再載入 VALUE 中。

(6)呼叫 PRTINT 或 PRTSGN 程式來輸出數目。

整個程式如下：

```
PINT     STA ASAVE        ;SAVE ACC
         STY YSAVE        ;SAVE Y REGISTER
         PLA              ;GET RETURN ADDRESS
         STA VALUE
         PLA
         STA VALUE+$1
         JSR INCV         ;INCREMENT VALUE BY TWO
         JSR INCV
         LDA VALUE+$1     ;PUSH RETURN ADDRESS
         PHA
         LDA VALUE
         PHA
         JSR DECV         ;MAKE VALUE POINT TO DATA
         JSR LVIV         ;GET DATA POINTED AT BY
                          ;DATA FOLLOWING JSR

         LDA ASAVE        ;RESTORE ACC
         LDY YSAVE        ;RESTORE Y REGISTER
         JMP PRTINT       ;CHANGE TO PRTSGN IF SIGNED
                          ;OUTPUT IS DESIRED
LVIV:
         JSR LAIA
         JSR LAIA
LAIA     LDY #$0
         LDA (VALUE),Y    ;GET L.O. BYTE
         PHA
         INY
         LDA (VALUE),Y    ;GET H.O. BYTE
         STA VALUE+$1     ;AND REPLACE VALUE
         PLA
         STA VALUE
         RTS
ASAVE    EPZ $4           ;ACC SAVE AREA
YSAVE    EPZ ASAVE+$1     ;Y REG!SAVE AREA
         END
```

數值輸入（*NUMERIC INPUT*）

16進位和 BCD（*HEXADECIMAL and BCD*）

數值輸入和數值輸出一樣的重要。這節中我們要討論多數值輸入的方法。

BCD 輸入是目前最容易作的，只需作一些罩幕和移動

的動作即可。BCD 輸入的方法如下：

(1)設定某位置（ VALUE ）的初值爲 0，下面例子都是採用 2 位元組的輸入，但可擴充到更多（或更少）位元組的輸入。

(2)假設所有輸入都存在第 2 頁（因此可與程式 GETLN 相通），且 Y 暫存器會指到要輸入的第一個字元。

(3) BCD 字串的結束被看成第一個碰到的非十進位數字。

(4)讀入每一個數字，然後用 4 個 ASL 指令把高階尼波（值永遠爲 $B ）移出去。所以原來數目的低階尼波就放在累積器的高階尼波上。

(5)用 ROL 指令把值移到 VALUE 。可用下面的程式：

```
; TSTDEC: TEST THE CHARACTER IN THE ACCUMULATOR.
; IF A VALID DECIMAL DIGIT, THEN THIS ROUTINE RETURNS
; WITH THE CARRY FLAG SET.  IF THE CHARACTER IN THE
; ACCUMULATOR IS NOT A DECIMAL DIGIT, THEN THIS ROUTINE
; RETURNS WITH THE CARRY FLAG CLEAR.
;
;
TSTDEC  CMP #"0"            ;BRACKET TEST FOR A DIGIT
        BLT NOTDEC
CMP #"9"+$1                 ;IS IT GREATER THAN NINE?
        BGE NOTDEC
        SEC                 ;IT IS A DECIMAL DIGIT
        RTS                 ;SO SET THE CARRY AND RETURN
;
NOTDEC  CLC                 ;NON-DIGIT WAS FOUND
        RTS
;
;
; SHFTIN: SHIFTS THE L.O. NIBBLE OF THE ACCUMULATOR
; INTO "VALUE"
;
SHFTIN  ASL                 ;MOVE LOW ORDER NIBBLE
        ASL                 ;INTO HIGH ORDER NIBBLE
        ASL                 ;OF THE ACCUMULATOR
        ASL
;
        JSR SHFT2
SHFT2   JSR SHFT1
SHFT1   ASL                 ;SHIFT ACC INTO VALUE
        ROL VALUE           ;NOTE: FOUR SHIFTS ARE
        ROL VALUE+$1        ;PERFORMED HERE!
        RTS
```

SHFTIN 的碼要仔細地看過。你可用筆來追踪從 JSR SHFT2 指令開始的碼，以確定確實作了 4 次的移動動作。利用這二個程式， BCD的輸入就變成很簡單了，如下：

```
;BCDIN: CONVERTS ASCII STRING IN PAGE TWO (POINTED
;AT BY THE Y REGISTER) INTO A BCD VALUE.  ALL DIGITS
;ARE CONVERTED UNTIL A NON-DIGIT IS ENCOUNTERED.
;
BCDIN:
        LDA #$0          ;INITIALIZE VALUE TO ZERO
        STA VALUE
        STA VALUE+$1
;
BCDLP   LDA PAG2,Y       ;GET NEXT CHARACTER
        JSR TSTDEC       ;IS IT A DECIMAL DIGIT?
        BCC BCDONE       ;IF NOT, QUIT
        JSR SHFTIN       ;IF IT IS, SHIFT INTO VALUE
        INY              ;INDEX TO NEXT CHARACTER
        BNE BCDLP        ;AND REPEAT

;
BCDONE  RTS
;
PAG2    EQU $200         ;GETLN INPUT BUFFER
VALUE   EPZ $2
        END
```

下個程式為在程式中利用 BCD 輸入程式的例子。

```
BCDTST  JSR PRINT        ;PRINT ROUTINE (SEE LAST CH)
        ASC "ENTER A NUMBER:"
        HEX 00
;
        JSR GETLN1       ;GET A LINE OF TEXT (NO PROMPT)
        LDY #$0
        JSR BCDIN
        JSR PRINT
        ASC "YOU ENTERED:"
        HEX 00
;
        LDA VALUE+$1
        JSR HEXOUT
        LDA VALUE
        JSR HEXOUT
        RTS
;
GETLN1  EQU $FD6F
HEXOUT  EQU $FDDA
        END
```

輸入16進位數目的方法也類似，只是用" TSTHEX "來替代" TSTDEC "測試累積器中的字是不是有效的16進位數 TSTHEX 不僅測試有效的16 進位數，同時也把字母" A "到" F "轉變成16進位值 $BA 到 $BF ，以使16進位值成爲連續的數值。

```
        CMP #"0"        ;BRACKET TEST FOR DECIMAL D
        BLT NOTHEX
        CMP #"9"+$1
        BLT ISHEX
        CMP #"A"        ;BRACKET TEST FOR "A" t
        BLT NOTHEX
        CMP #"G"
        BGE NOTHEX      ;SAME AS BCS (SEE NEXT INSTR)
        SBC #$6         ;CONVERT FROM $C1 TO $BA ...
ISHEX   SEC             ;SIGNAL VALID HEX DIGIT
        RTS
;
NOTHEX  CLC             ;SIGNAL INVALID HEX DIGIT
        RTS
```

要輸入16進位數目就在 BCDIN 程式中用這程式來代替 TSTDEC ，且用 JSR TSTDEC 來代替 JSR TSTHEX 。當然要改一下程式的名稱，使其更有意義。

不帶符號十進位的輸入 (*UNSIGNED DECIMAL INPUT*)

十進位的輸入只比 BCD 或16進位數目的輸入稍微困難一點。方法如下：

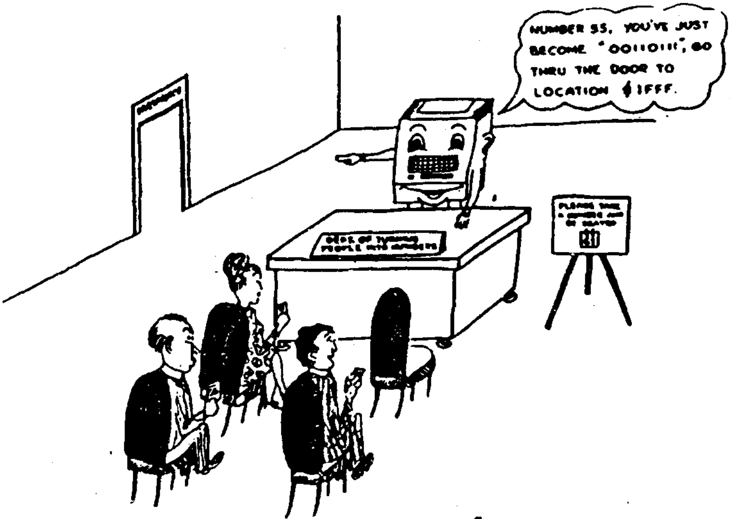
(1)輸入一字元且測試是否有效（也就是說是否在0～9之間）。

(2)把高階位元組拿掉，變成爲數字的數目表示法。

(3)將16位元位置的值乘以10再加上(2)中所得的數字。

(4)當所有的數字都移入之後，那兩個位置上所存的值就

是輸入的十進位值的二進位表示法。



"DECIMAL INPUT"

第 1 ，2 步驟和 BCD 輸入的方式一樣，第 3 步驟可用下章要討論的乘法來作，但用較特定的乘法程式會更快且碼會更少。下列程式把位置 VALUE 和 VALUE+$1 的值乘 10 ：

```
MUL100
        PHP
        PHA
        ASL VALUE        ;MULTIPLY VALUE BY 2
        ROL VALUE+$1
        LDA VALUE+$1     ;SAVE A COPY OF VALUE
        PHA              ;MULTIPLIED BY 2
        LDA VALUE
        ASL VALUE        ;NOW MULTIPLY VALUE BY 8
        ROL VALUE+$1     ;SINCE VALUE HAS ALREADY
        ASL VALUE        ;BEEN MULTIPLIED BY 2
        ROL VALUE+$1     ;A SIMPLE MULTIPLY BY 4 GIVES
        CLC
        ADC VALUE        ;ADD IN 2×VALUE TO 8×VALUE
        STA VALUE        ;TO OBTAIN 10×VALUE
        PLA
        ADC VALUE+$1
        STA VALUE+$1
        PLA
        PLP
        RTS
```

每次呼叫這個程式就會把位置 VALUE 的值乘以 10 ，而不影響所有的暫存器。

最後的步驟（把數字加到 16 位元數目上）就很簡單了！整個十進位輸入的程式如下：

```
; DECIMAL INPUT ROUTINE
;
; NOTE: THIS ROUTINE ASSUMES THAT GETLN HAS BEEN
;       CALLED AND THAT THE X-REGISTER POINTS TO
;       THE FIRST VALID DECIMAL DIGIT IN PAGE 2.
;       UPON EXIT, THE X-REGISTER POINTS TO  THE
;       FIRST NON-DIGIT ENCOUNTERED.
;
;
DECINP:
        PHP             ;SAVE STATUS
        PHA             ;AND ACC
;
        LDA #$0         ;INITIALIZE VALUE
        STA VALUE       ;TO ZERO
        STA VALUE+$1
;
DECLP   LDA INPUT,X     ;GET THE NEXT DIGIT
        JSR TSTDEC      ;IS IT REALLY A DIGIT?
        BCC ALDONE      ;IF NOT, QUIT
        AND #$F         ;OTHERWISE CONVERT TO A NUMBER
        JSR MULT10      ;MULTIPLY VALUE BY 10
        CLC
        ADC VALUE       ;AND ADD IN CURRENT DIGIT
        STA VALUE
        BCC DECLP       ;IF NOT CARRY, LOOP BACK
        INC VALUE+$1    ;IF A CARRY EXISTS, ADD ONE
        JMP DECLP       ;TO VALUE+$1 AND LOOP BACK
;
ALDONE  PLA             ;RESTORE REGISTERS
        PLP
        RTS
;
;
; TSTDEC: TEST ACC TO SEE IF IT IS A VALID DECIMAL
;         DIGIT.  IF SO, THE CARRY FLAG IS SET.
;         OTHERWISE THE CARRY FLAG IS CLEAR.
;
TSTDEC:
        CMP #"0"
        BLT NOTDEC
        CMP #"9"+$1
        BGE NOTDEC
        SEC
        RTS
;
NOTDEC  CLC
```

```
RTS
;
;
; MULT 100 MULTIPLIES VALUE BY TEN
;        (SEE ABOVE)
;
MULT100
        PHP
        PHA
        ASL VALUE
        ROL VALUE+$1
        LDA VALUE+$1
        PHA
        LDA VALUE
        ASL VALUE
        ROL VALUE+$1
        ASL VALUE
        ROL VALUE+$1
        CLC
        ADC VALUE
        STA VALUE
        PLA
        ADC VALUE+$1
        STA VALUE+$1
        PLA
        PLP
        RTS
;
; THAT'S ALL FOLKS...
```

這程式有幾個缺點。第一，它並沒有測試溢位的情況。第二，碰到第一個非數字的字元就立刻停止，也就是說壞的資料不會被查覺。最後若第一個碰到的字不是十進位數則程式立刻停止且送回零值。

這三個問題都可以解決。在 MULT10 程式中，每次作完 ROL 指令之後和在 DECINP 程式中每次作完加法之後就測試進位旗標看是否爲 1 ，如此就可測試溢位了。

第二個問題會使得不合法的資料不被查覺出來。通常數值的輸入可用空白， return 或，（或其他任何特殊符號）來作結束，若不是碰到這些特殊符號中的某一個字就表示發生輸入錯誤。因此這問題的解決方法是檢查第一個非數字字元，看是否爲合法的界限字（ delimiters ）。

最後一個問題（無效的第一個字）是第二個問題的擴充。解決方法很簡單：首先刪去所有開頭的空白（因此數目中允許有開頭的空白）。其次測試第一個不是空白的字，看是否爲有效的十進位數字，若不是則報告錯誤。下列程式即考慮了這些問題，同時或多或少模擬了APPLE整數BASIC中的整數輸入：

```
DECINP:
        PHP
        PHA
;
DOIT:
        LDA #$0          ;INIT VALUE
        STA VALUE
        STA VALUE+$1
        JSR BLKDEL       ;DELETE LEADING BLANKS
        JSR TSTDEC       ;IS FIRST NON-BLANK A DIGIT?
        BCC BADDIG       ;IF NOT, INFORM THE USER
;
DECLP   LDA INPUT,X      ;GET NEXT (OR FIRST) DIGIT
        INX              ;MOVE TO NEXT CHARACTER
        JSR TSTDEC       ;IS IT A DIGIT?
        BCC ALDONE       ;IF NOT, QUIT
        AND #$F          ;CONVERT TO A NUMBER
        JSR MULT10       ;MULTIPLY VALUE BY 10
        BVS OVRFLW       ;IF OVERFLOW, INFORM USER
        CLC

        ADC VALUE        ;ADD CURRENT DIGIT
        STA VALUE        ;TO VALUE
        BCC DECLP
        INC VALUE+$1     ;IF CARRY, INCREMENT VALUE+
        BNE DECLP        ;IF NO OVERFLOW, LOOP BACK
        JMP OVRFLW       ;IF OVRFLOW, INFORM USER
;
ALDONE  CMP #","         ;TEST FOR VALID DIGIT
        BEQ QUIT         ;DELIMITERS
        CMP #" "
        BEQ QUIT
        CMP #$8D         ;RETURN IS VALID
        BEQ QUIT
        JSR PRINT        ;PRINT ROUTINE FROM A PREVIOUS
        HEX 8D           ;CHAPTER
        ASC "RETYPE NUMBER"
        HEX 8D00
        JSR GETLN        ;READ A LINE OF TEXT
        LDX #$0
        JMP DOIT
```

```
;
OVRFLW   JSR PRINT
         HEX 8D
         ASC ">65535"
         HEX 8D00
         JSR GETLN          ;GET A NEW LINE OF TEXT
         LDX #$0
         JMP DOIT
;
;
QUIT:
         PLA
         PLP
         RTS
;
;
; BLANK DELETION ROUTINE
;
BLKDEL   LDA INPUT,X
         CMP #" "
         BNE BLKD1
         INX
         BNE BLKDEL
;
BLKD1    RTS
;
; MULTIPLY BY 10 ROUTINE
;
MULT10   PHA                ;CAN'T SAVE CARRY, V IS OVRFLW
;
         ASL VALUE
         ROL VALUE+$1
         BCS MOVRFL
         LDA VALUE+$1
         PHA
         LDA VALUE
         ASL VALUE
         ROL VALUE+$1
         BCS MOVRFL
         ASL VALUE
         ROL VALUE+$1
         BCS MOVRFL
         CLC
         ADC VALUE
         STA VALUE
         PLA
         ADC VALUE+$1
         STA VALUE+$1
         BCS MOVRFL
         PLA
         BIT NOVRFL         ;SET V FLAG TO ZERO
         RTS

MOVRFL   BIT OVERFL         ;SET V FLAG TO ONE
         RTS
```

```
;
NOVRFL  HEX 00
OVERFL  HEX 40
;
;
; TSTDEC: TESTS CHARACTER IN ACC TO SEE IF IT IS
;         A VALID DECIMAL DIGIT
;         CARRY IS SET IF IT IS
;
TSTDEC:
        CMP #"0"
        BLT NOTDEC
        CMP #"9"+$1
        BGE NOTDEC
        SEC
        RTS
;
NOTDEC  CLC
        RTS
;
;
INPUT   EQU $200
VALUE   EPP $0
;
GETLN   EQU $FD67
;
;
; NOTE: THE PRINT ROUTINE PROVIDED IN THE PREVIOUS
;       CHAPTER MUST BE INCLUDED HERE
;
;
;
```

用這個程式，先呼叫 GETLN 來讀入一行資料，再令X暫存器指到要輸入的十進位數字的位址，然後 JSR DECINP。從 DECINP 跳回時，所要的數目（二進位型式）就存在位置 VALUE 和 VALUE+$1 上。或許你希望做某些改進的工作，如：

(1)使它能處理三，四（或更多）位元組的整數。

(2)能在 JSR DECINP 之後直接指明位址，讓輸入資料的值存在該位址上。

帶符號十進位的輸入
（*SIGNED DECIMAL INPUT*）

既然有了不帶符號十進位輸入的程式，帶符號十進位的輸入就變成很容易了。只要檢查第一個非空白字元是否爲負號即可。若是負號，則加到下面字元，然後呼叫不帶符號十進位輸入的程式。當從程式跳回時，檢查位置 VALUE+$1 的位元 7，若爲 1 則表示發生溢位；否則若爲負數，則取位置 VALUE 和 VALUE+$1 上的值的 2 的補數值，若爲正數則值不變。程式如下：

```
; SIGNED DECIMAL INPUT
;
SNGDEC:
         PHP
         PHA
;
DOSGN:
         JSR BLKDEL
         CMP #"-"
         BNE SGN1
         LDA #$1           ;SET A FLAG SIGNIFYING
         STA SIGN          ;A MINUS VALUE
         INX
         JMP SGN2
;
SGN1     LDA #$0           ;SET A FLAG SIGNIFYING
         STA SIGN          ;A POSITIVE NUMBER
;
SGN2     JSR DECINP        ;GET THE UNSIGNED NUMBER
         LDA VALUE+$1      ;TEST FOR OVERFLOW
         BMI SGNOVR
         LDA SIGN          ;TEST TO SEE IF 2'S COMP
         BFL DONE          ;IS REQUIRED
         SEC               ;PERFORM 2'S COMP
         LDA #$0           ;OPERATION
         SBC VALUE
         STA VALUE
         LDA #$0
         SBC VALUE+$1
;
DONE     PLA
         PLP
         RTS
;
SGNOVR   JSR PRINT
         HEX 8D
         ASC ">32767  REENTER"
         HEX 8D00
         JSR GETLN
         LDX #$0
         JMP DOSGN
;
SIGN     EPZ VALUE+$2
```

這完成了正常的類似 BASIC 中的數值 I/O 的運算。這裡列出的程式都是基本的程式，將其修改就可作多位元組的輸入，單位元組輸入等 I/O 運算。其餘的數值輸入，如二進位或八進位，可由修改 MULT10 和 TSTDEC 程式來完成。事實上，應該要寫成一個可輸入任何底數的資料，但首先要看下一章的乘法程式。

第十三章

乘法和除法

概　論（GENERAL）

前面提過6502 並沒有乘法，和除法的指令。當然若有乘除法的指令就很方便，所以我們要用副程式來提供乘除的能力。

乘法（MULTIPLICATION）

在二進位中，乘法非常的簡單。事實上，是和十進位的乘法類似。看看下面十進位乘法的例子：

```
   10110
×    110
 -------
```

把（ 0 × 10110 ）+（ 10 × 10110 ）+（ 100 × 10110 ）就是結果。乘以 10 尤其簡單：只要把數目往左移一位即可。因此上式的結果爲 1112100 。

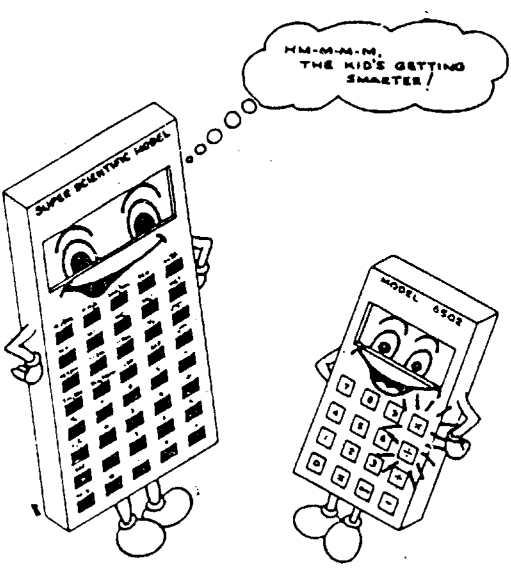
二個二進位數的乘法也是同樣的道理。把（ 0 × 10110 ）+（ 10 × 10110 ）+（ 100 × 10110 ）就可得到結果。

乘以 2 的乘冪時，只要用 ASL 或 ROL 指令來作移位即可。上式的答案爲 10000100 。

n 位元乘以 m 位元的結果最多爲（ m + n ）位元。因此 8 位元乘以 8 位元會產生最多 16 位元的結果。同理 16 位元相乘會產生最多 32 位元的結果。記住這點，以便在作乘法時能確定有足夠的位置來存放結果。

下面的乘法程式用了第 0 頁中的六個位置，分別來存乘數，被乘數和部分結果。這些位置（都是 16 位元）分別標示成 MULPLR , MULCND 和 PARTIAL 。作完乘法之後所得結果的低階 16 位元就放在位置 MULPLR 和 MULPLR + $1 上，而高階的 16 位元則在位置 PARTIAL 和 PARTIAL + $1 上。公式如下：

(MULPLR, PARTIAL) = MULPLR × MULCND + PARTIAL



LEARNING MULTIPLICATION AND DIVISION.

注意到 PARTIAL 的值被加到乘法的結果上。這在許多數學計算，包括擴充精確度的乘法時非常有用。在這裡，只要記得在呼叫乘法程式之前先把位置 PARTIAL 和 PARTIAL +$1 的值設爲 0 就可以了！

下面的乘法程式是修改 Apple 監督程式位置 $FB63 上的乘法程式所得的結果。

```
; USMUL- UNSIGNED 16-BIT MULTIPLICATION
;        32 BIT RESULT IS RETURNED IN LOCATIONS
;        (MULPLR, PARTIAL).
;
;
USMUL:
        PHA
        TYA
        PHA
;
USMUL1  LDY #$10          ;SET UP FOR 16-BIT MULTIPLY
USMUL2  LDA MULPLR        ;TEST L.O. BIT TO SEE IF SET
        LSR
        BCC USMUL4
;
        CLC               ;L.O. BIT SET, ADD MULCND TO
        LDA PARTIAL       ;PARTIAL PRODUCT
        ADC MULCND
        STA PARTIAL
        LDA PARTIAL+$1
        ADC MULCND+$1
        STA PARTIAL+$1
;
; SHIFT RESULT INTO MULPLR AND GET THE NEXT BIT
; OF THE MULTIPLIER INTO THE LOW!ORDER BIT OF
; MULPLR
;
USMUL4  ROR PARTIAL+$1
        ROR PARTIAL
        ROR MULPLR+$1
        ROR MULPLR
;
; SEE IF DONE YET
;
        DEY
        BNE MUL2
        PLA
        TAY
        PLA
        RTS
;
;
MULPLR  EPZ $50
PARTIAL EPZ MULPLR+$2
MULCND  EPZ PARTIAL+$2
```

下面爲使用乘法函數的例子：

```
; COMPUTE 25 × 66 AND LEAVE RESULT IN
; "RESULT"
;
EXMPL:
        LDA #!25          ;25 DECIMAL
        STA MULPLR
        LDA /!25          ;H.O. BYTE OF 25
        STA MULPLR+$1
        LDA #!66
        STA MULCND
        LDA /!66
        STA MULCND+$1
        LDA #$0           ;MUST SET PARTIAL TO ZERO
        STA PARTIAL
        STA PARTIAL+$1
        JSR USMUL         ;PERFORM THE MULTIPLICATION
        LDA MULPLR        ;MOVE PRODUCT TO RESULT
        STA RESULT
        LDA MULPLR+$1
        STA RESULT+$1


        ETC...
```

假如作 16 位元× 16位元的乘法而把結果存在一個 16 位元的位置上，則需用 PARTIAL OR PARTIAL + $ 1 指令來測試溢位。假如結果不是 0，則表示發生溢位。

前面提過，可用 PARTIAL 來擴充這程式使它可作 24, 32, 48 或 64 個位元的乘法。這很簡單只要稍稍修改原來的程式即可。修改如下：把要相乘的位元數目載入 Y 暫存器，然後修改多重精確度的 ROR 串列和 ADC 串列使合乎你所選擇的精確度。同時不要忘記要留更多的位置給 MULCND ， MULPLR 和 PARTIAL ！下面爲 24 × 24 位元的乘法：

```
; USMUL- UNSIGNED 24-BIT MULTIPLICATION
;        48 BIT RESULT IS RETURNED IN LOCATIONS
;        (MULPLR, PARTIAL)
;
;
USMUL:
        PHA
        TYA
        PHA
```

```
;
USMUL1  LDY #$18          ;SET UP FOR 24-BIT MULTIPLY
USMUL2  LDA MULPLR        ;TEST L.O. BIT TO SEE IF SET
        LSR
        BCC USMUL4
;
        CLC               ;L.O. BIT SET, ADD MULCND TO
        LDA PARTIAL       ;PARTIAL PRODUCT
        ADC MULCND
        STA PARTIAL
        LDA PARTIAL+$1
        ADC MULCND+$1
        STA PARTIAL+$1
        LDA PARTIAL+$2
        ADC MULCND+$2
        STA PARTIAL+$2
;
; SHIFT RESULT INTO MULPLR AND GET THE NEXT BIT
; OF THE MULTIPLIER INTO THE LOW ORDER BIT OF
; MULPLR
;
USMUL4  ROR PARTIAL+$2
        ROR PARTIAL+$1
        ROR PARTIAL
        ROR MULPLR+$2
        ROR MULPLR+$1
        ROR MULPLR
;
; SEE IF DONE YET
;
        DEY
        BNE MUL2
        PLA
        TAY
        PLA
        RTS
;
;
MULPLR  EPZ $50
PARTIAL EPZ MULPLR+$3
MULCND  EPZ PARTIAL+$3
```

該注意上面的程式只作不帶符號的乘法而已。若作帶符號的乘法，則首先要看乘數和被乘數的符號是否一樣，並且使用一個旗標來記錄相同的情形。然後取乘數和被乘數的絕對值，再由不帶符號的乘法。最後再來測試符號旗標，若旗標表示二數的符號不一樣，則結果應爲負數。

```
; SIGNED 16-BIT MULTIPLICATION
;
SMUL:
        PHA
        TYA
        PHA

        LDA MULCND+$1     ;TEST SIGN BITS
        XOR MULPLR+$1     ;TO SEE IF H.O BITS ARE UNEQU
        AND #$80
        STA SIGN          ;SAVE SIGN STATUS
        JSR ABS1          ;TAKE ABSOLUTE VALUE OF MULPLR
        JSR ABS2          ;TAKE ABSOLUTE VALUE!OF MULCND
        JSR USMUL         ;UNSIGNED MULTIPLY
        LDA SIGN          ;TEST SIGN FLAG
        BPL SMUL1         ;IF NOT SET, RESULT IS CORRECT
        JSR NEGATE        ;NEGATE RESULT
;
SMUL1   PLA
        TAY
        PLA
        RTS
;
;
ABS1    LDA MULPLR+$1     ;SEE IF NEGATIVE
        BPL ABS12
;
NEGATE:
        SEC               ;NEGATE MULPLR
        LDA #$0
        SBC MULPLR
        STA MULPLR
        LDA #$0
        SBC MULPLR+$1
        STA MULPLR+$1
;
ABS12   RTS
;
;
ABS2    LDA MULCND+$1     ;SEE IF NEGATIVE
        BPL ABS22
        SEC               ;NEGATE MULCND
        LDA #$0
        SBC MULCND
        STA MULCND
        LDA #$0
        SBC MULCND+$1
        STA MULCND+$1
;
ABS22   RTS
```

跟不帶符號的乘法一樣，可用指令 PARTIAL OR PARTIAL+$1 來測試溢位。在較早的 Apple 監督程式中，提供了一個帶符號乘法的程式。讀者應該研究一下它的

乘法程式，因爲它用的技巧和這裡所用的技巧大不相同。它比較複雜並且難懂，但是可以做爲如何犧牲速度和可讀性換取更少的程式碼的練習。

除法演算法
（*DIVISION ALGORITHMS*）

跟乘法一樣，二進位的除法和一般人們作長除法時的方法類似。首先看除數的高階位元，若爲1再看看被除數是否可除，則記上商數且把被除數減去目前的除數。反覆這步驟直到作完所有數字（或位元）爲止。除法的程式如下：

```
; UNSIGNED 16-BIT DIVISION
; COMPUTES (DIVEND,PARTIAL) / DIVSOR
; (I.E., 32 BITS DIVIDED BY 16 BITS)
;
USDIV:
        PHA
        TYA
        PHA
        TXA
        PHA
;
        LDY #$10          ;SET UP FOR 16 BITS
USDIV2  ASL DIVEND
        ROL DIVEND+$1
        ROL PARTIAL
        ROL PARTIAL+$1
        SEC               ;LEAVE DIVEND MOD DIVSOR
        LDA PARTIAL       ;IN PARTIAL
        SBC DIVSOR
        TAX
        LDA PARTIAL+$1
        SBC DIVSOR+$1
        BCC USDIV3
;
        STX PARTIAL
        STA PARTIAL+$1
        INC DIVEND
;
USDIV3  DEY
        BNE USDIV2
```

```
;
            PLA
            TAX
            PLA
            TAY
            PLA
            RTS
;
;
DIVEND   EPZ $50
PARTIAL  EPZ DIVEND+$2
DIVSOR   EPZ PARTIAL+$2
```

值得一提的是這個程式同時也算出 DIVEND MOD DIVSOR 的值，結果存在位置 PARTIAL 上。若除數爲 0，則 DIVEND 會變成 $FFFF 。你的程式可用 DIVEND AND DIVEND+$1 的指令然後比較結果是否爲 $FF 來測出這問題。現在還有一個問題： $FFFF 除以 1，結果也是 $FFFF 。解決方法很簡單，只要在呼叫 USDIV 之前，先測試被除數是否爲 $FFFF 即可。擴充除法程式成可適用於任何數目的位元組，則把所要的精確度位元數載入 Y 暫存器中，再增加幾個 ROL 指令，並且在 SBC 指令上擴充其精確度即可。

使用這個程式時，把 32 位元的除數載入位置 DIVEND， DIVEND+$1， PARTIAL 和 PARTIAL+$1 上（低階位元組在位置 DIVEND，而高階位元組在 PARTIAL + $1 上），再把 16 位元的除數載入 DIVSOR 。接著 JSR USDIV 。若你的被除數只有 16 位元，則把 0 存到 PARTIAL 和 PARTIAL+$1 即可。

```
; EXAMPLE: DIVIDE 195 BY 24 AND PUT THE QUOTIENT
;          INTO "RESULT"
;          STORE THE MODULO OF 195/24 IN LOCATION
;          "MODULO"
;
EXMPL:
          LDA #!195        ;DECIMAL 195
          STA DIVEND
          LDA /!195
          STA DIVEND+$1
          LDA #!24         ;DECIMAL 24
          STA DIVSOR
          LDA /!24
          STA DIVSOR+$1
          LDA #$0          ;PERFORMING A 16 BY 16 DIVISION
          STA PARTIAL
          STA PARTIAL
          JSR USDIV
          LDA DIVEND
          STA RESULT
          LDA DIVEND+$1
          STA RESULT+$1
          LDA PARTIAL
          STA MODULO
          LDA PARTIAL+$1
          STA MODULO+$1
              .
          ETC ...
```

帶符號除法比不帶符號除法要麻煩一些。和帶符號乘法一樣，得使用一個符號旗標來決定最後商數的符號。同理也是取除數和被除數的絕對值來呼叫不帶符號除法程式，最後若符號旗標爲 1 ，則商數取負數，否則商數取正值。

若除數爲 0 ，則商數爲 $FFFF 。用不帶符號除法的程式只有當 $FFFF 除以 1 時才會得 $FFFF ，但用帶符號除法的程式，則 $FFFF 除以 1 ，除以 $FFFF 都會得 $FFF（爲－1 的表示法），除以 0 的結果也是 $FFFF ，這麼多情況會搞不清楚，因此在下列的帶符號除法程式中，用溢位旗標來表示除數是否爲 0 ，若除數爲 0 ，則溢位旗標爲 1 ，否則爲 0 。因此在從除法程式中跳回時，你就可檢查溢位旗標而作適當的措施。也可用這技巧在不帶符號除法程式中來

處理$FFFF 除以 1 的情況。

```
; SIGNED 16-BIT DIVISION ROUTINE
; V FLAG IS RETURNED SET IF ZERO DIVIDE OCCURS
;
; THIS ROUTINE COMPUTES (DIVEND.PARTIAL)/DIVEND
; AS WELL AS (DIVEND.PARTIAL) MOD DIVEND
;
SDIV:
        PHA
;
        LDA DIVEND+$1     ;CHECK SIGN BITS
        XOR DIVSOR+$1
        AND #$80
        STA SIGN
        JSR DABS1         ;ABSOLUTE VALUE OF DIVSOR
        JSR DABS2         ;ABSOLUTE VALUE OF DIVEND
        JSR USDIV         ;COMPUTE UNSIGNED DIVISION
        LDA DIVEND        ;CHECK FOR ZERO DIVIDE
        AND DIVEND+$1
        CMP #$FF
        BEQ OVRFLW
        LDA SIGN          ;SIGN IF RESULT MUST BE
        BPL SDIV1         ;NEGATIVE
        JSR DIVNEG
;
SDIV1   CLV               ;NO ZERO DIVISION
        PLA
        RTS
;
OVRFLW  BIT SETOVR        ;SET OVERFLOW FLAG
        PLA
        RTS
;
SETOVR  HEX 40
;
;
DABS1   LDA DIVSOR+$1     ;SEE IF NEGATIVE
        BPL DABS12
;
DIVNEG:
        SEC               ;NEGATE DIVSOR
        LDA #$0
        SBC DIVSOR
        STA DIVSOR
        LDA #$0
        SBC DIVSOR+$1
        STA DIVSOR+$1
```

```
;
DABS12  RTS
;
;
DABS2   LDA DIVEND+$1     ;SEE IF NEGATIVE
        BPL DABS22
        SEC               ;NEGATE DIVEND
        LDA #$0
        SBC DIVEND
        STA DIVEND
        LDA #$0
        SBC DIVEND+$1
        STA DIVEND+$1
;
DABS22  RTS
```

本章所列的程式都可輕易地修改，使它適用於特殊的情況。在許多例子中尤其在擴充到較 16 位元更多運算的時候，若用廻路會減少不少碼，但速度的因素卻更重要。

第十四章

處理字串的運算

字串的處理　（*STRING HANDLING*）

在計算機中有許多表示字串的方法，但不論如何表示，字串都有三種特性：(1)有最大長度，即分配給它的位元組數。(2)有動機的執行時長度（ "run-time" length ），即目前字串所用的位元組數。(3)有在記憶體中的起始位址。

這裡將採用適於 LISA 結構的用法。以後提到的字串都是下列三種形式之一：

(1)字串由一群從某特定位置開始的字元組成，且由某特定值，如 $00 作爲結束。（在前幾章的 PRINT 程式中用過）。

(2)字串由一群某已知位置開始的字元組成，而由一字元作爲結束，此字元的高階位元和字串的其他字元的高階位元相反。

(3)字串由一長度位元組和一串該長度的字元所組成。

前二種形式主要在輸出入時有用。$00 常用來作輸出字元的結束符號，而 $8D 則用來作爲輸入字元的結束符號（因爲在大部分情形，都是用 carriage return 作爲結束）。第二種形式是第一種型式的特殊例子。在最後一字元的高階位

元作記錄，所以就不必在後面多加一個位元組了。要注意，第二種形式只能有128個不同字元，而不像第一種形式有256種字元，但對每一字串而言都節省了一個位元組。LISA有一個特殊的假指令使用這種形式儲存字串，這指令為" DCI ，它把字串存在記憶體中，並且令最後一字的高階位元與其他字不同。

第三種形式是最常用的，因為它最方便。採用這種形式之後，一些有關字串的函數，如串聯，長度和次字串都變成很簡單。本章所列的字串處理程式都是採用這種形式。但由於在程式中也可能有第1，2種形式的字串，所以要有把第1種和第2種形式轉變成第三種形式的轉變程式。

把第一種轉換成第三種時，需要有3種資料。第一為第一種形式字串的起始位置；第二為第三種形式字串的起始位置，即轉變後的字串的存放位置；第三為第一種形式字串的結束符號。在我們的程式中，假設這三種資料放在位置TYPE1，TYPE3和DLMTR上。TYPE1和TYPE3都是16位元的位址，每個都在第0頁中佔二個位元組的位置。而DLMTR則只佔一位元組。在呼叫我們的程式之前，先要把這三種資料設定好。

這程式會從TYPE1所指到的位置中取出一個字元，然後存在TYPE3所指的相對位置上，當然作了些小改變。因為TYPE3所指的第一個位元組應為字串的長度，所以在把字元存到指定位置之前要把TYPE3所指位置的值加1。最後字元都轉移好了之後，字串的長度要放在第一個位置上。程式如下：

```
; TYPE1 TO TYPE3 STRING CONVERSIONS
;
; POINTERS TO THE RESPECTIVE DATA AREAS
; ARE PASSED IN "TYPE1" AND "TYPE2"
; "DLMTR" CONTAINS THE STRING DELIMITER BEING USED
;
TYPE1    EPZ $0
TYPE3    EPZ TYPE1+$2
DLMTR    EPZ TYPE3+$2
;
;
T1T03:
;
         PHP                  ;SAVE ALL THE REGISTERS
         PHA
         TYA
         PHA
;
         INC TYPE3            ;ADD ONE TO TYPE3 POINTER
         BNE T1T03A           ;SO THAT IT POINTS TO THE FIRST
         INC TYPE3+$1         ;AVAILABLE CHAR PAST THE LENGTH
;
T1T03A:
         LDY #$0              ;SET UP INDEX TO ZERO
T1T03B   LDA (TYPE1),Y        ;FETCH TYPE1 CHARACTER
         CMP DLMTR            ;IS IT THE DELIMITER?
         BEQ T1T03C           ;IF SO, PREPARE TO QUIT
         STA (TYPE3),Y        ;OTHERWISE TRANSFER
         INY                  ;MOVE TO NEXT CHARACTER
         BNE T1T03B           ;DON'T ALLOW STRINGS > 255
         DEY                  ;IF OVERFLOW OCCURS, TRUNCATE
;
;
T1T03C   LDA TYPE3            ;DECREMEOT TYPE3 POINTER SO
         BNE T1T03D           ;IT POINTS TO LENGTH BYTE AGAIN
         DEC TYPE3+$1
T1T03D   DEC TYPE3
;
         TYA                  ;TRANFER LENGTH OF STRING TO A
         LDY #$0              ;SET UP INDEX TO LENGTH BYTE
         STA (TYPE3),Y        ;STORE LENGTH IN FIRST BYTE
;
         PLA                  ;RESTORE THE REGISTERS
         TAY
         PLA
         PLP
         RTS
```

假如用程式 GETLNZ 從 Apple 鍵盤上讀入一行文字，則可將其轉換成第三種形式的字串，程式如下：

```
          LDA #$200        ;INIT TYPE1 TO $200
          STA TYPE1
          LDA /$200
          STA TYPE1+$1
;
          LDA #STRING      ;PUT ADDRESS OF DESTINATION
          STA TYPE3        ;STRING INTO "TYPE3"
          LDA /STRING
          STA TYPE3+$1
;
          LDA #$8D         ;INITILIZE THE DELIMITER
          STA DLMTR        ;CHARACTER TO RETURN
          JSR T1T03        ;PERFORM THE CONVERSION


          ETC.
```

第二種形式也可以同樣方式轉換成第三種型式，程式如下：

```
;TYPE 2 TO TYPE 3 STRING CONVERSION
;TYPE 2 STRING IS ASSUMED TO BE A STRING WHOSE HIGH
;ORDER BITS ARE ALL SET EXCEPT FOR THE LAST CHARACTER
;WHOSE HIGH ORDER BIT IS CLEAR
;THIS CAN BE MODIFIED BY REPLACING THE "BPL"
;INSTRUCTION WITH A "BMI" IF DESIRED
;
TYPE2     EPZ $0
TYPE3     EPZ TYPE2+$2
;
;
T2T03:
          PHP              ;SAVE THE REGISTERS
          PHA
          TYA
          PHA
;
          INC TYPE3        ;MOVE PAST THE LENGTH BYTE
          BNE T2T03A
          INC TYPE3+$1
;
T2T03A:
          LDY #$0          ;INITIALIZE STRING INDEX
T2T03B    LDA (TYPE2),Y
          BPL T2T03C
          STA (TYPE3),Y
          INY
          BNE T2T03B       ;PREVENT OVERFLOW
          DEY              ;TRUCATE TO 255 CHARS
;
T2T03C    ORA #$80         ;STORE LAST CHARACTER
          STA (TYPE3),Y
          INY              ;ADJUST LENGTH
          BNE T2T03D       ;TEST FOR OVERFLOW
          DEY              ;TRUCATE IF > 255 CHARS
```

```
;
T2T03D  LDA TYPE3        ;MOVE TYPE3 POINTER BACK
        BNE T2T03D       ;LENGTH BYTE
        DEC TYPE3+$1
T2T03E  DEC TYPE3
;
        PLA              ;RESTORE THE REGISTERS
        TAY
        PLA
        PLP
        RTS
```

反方向的轉換（從第三種到第一或第二種）較少用，但也很簡單。這留給讀者作爲習題。

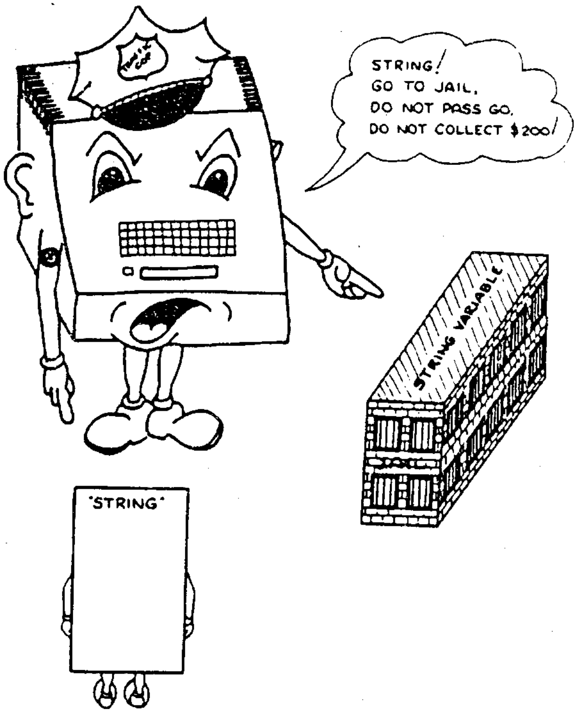
宣告字串常數
（*DECLARING LITERAL STRINGS*）

除了從鍵盤上輸入字串之外，有時候也要在程式中設定一些字串。LISA 提供了一個假指令“ STR ”可幫你作這工作。這指令輸出一由 ASCII 字元組成的字串，且在最前面加上其長度。STR 在宣告字串常數時候很有用。在用 STR 指令宣告字串常數時，要記住用雙引號把字串括起來（不用單引號）。

字串設定　（*STRING ASSIGNMENTS*）

最基本且有用的字串運算應該是字串設定。最簡單的字串設定只是把資料從某一位置移到另一位置而已。假設要把

字串從位置“ STR1 ”移到位置“ STR2 ”，則程式如下：



"STRING ASSIGNMENTS"

```
        LDY STR1        ;GET THE LENGTH BYTE
LOOP    LDA STR1,Y      ;TRANSFER STRING
        STA STR2,Y
        DEY
        BNE LOOP
        LDA STR1        ;TRANSFER THE LENGTH OVER
        STA STR2
```

你可看出第 1 位元組到第 n 位元組的資料被移動，然後 STR1 字串的長度存在位置 STR2 上。

若在程式中常作字串指定的動作，則最好是把它寫成副程式，在 JSR 之後來指明二個字串的位址，以便多次呼叫，如下例：

```
JSR SASIGN        ;STRING ASSIGNMENT
ADR DEST          ;DEST = SOURCE
ADR SOURCE
```

這麼一來每次設定字串，只需要 7 個位元組即可。另一種特殊例子的寫法如下：

```
JSR SASGMI        ;IMMEDIATE STRING ASSIGNMENT
ADR DEST          ;ADDRESS OF STRING
STR "HELLO"       ;STRING TO BE ASSIGNED
```

這種寫法允許你使用簡單的方法設定某字串常數。以上二種方法的程式留給讀者作爲習題，可參考前幾章所講的輸出程式。

字串函數（*STRING FUNCTIONS*）

最基本的字串函數是長度函數。它是以下要提到的字串函數的基礎。作法很簡單，因爲字串的第一個位元組即爲其長度，所以長度函數只是載入指令而已。如下面的字串宣告:

```
STRING   STR   "HELLO THERE"
```

然後用　LDA STRING　指令就把字串的長度載入累積器中。

輸出字串則很簡單，下面程式輸出位置' STRING ' 上的字串：

```
         LDA STRING         ;CHECK LENGTH TO INSURE
         BEQ XIT            ;IT IS NOT ZERO
;
         LDY #$0            ;SET UP INDEX TO FIRST CHAR
LOOP     LDA STRING+$1,Y    ;GET THE NEXT CHARACTER
         JSR COUT           ;OUTPUT IT
         INY
         CPY STRING         ;DONE YET?
         BLT LOOP
```

0 載入 Y 暫存器而位置 STRING + 1 的值則載入累積器中。如此一來當指標指到字串最後字元的後一個字元時，Y 暫存器就等於字串的長度。也就是當存有有效索引時，Y 暫存器的值永遠小於字串的長度。這使得我們可用使用 BLT 指令作爲廻路的結束。

另外一個比較好的字串輸出程式是在 JSR 之後列出所要輸出字串的位址，和前面的輸出程式類似。程式如下：

```
PRTSTR:
         STA ASAVE
         STY YSAVE
         PLA                ;GET RETURN ADDRESS FROM
         STA RTNADR         ;THE 6502 STACK
         PLA
         STA RTNADR+$1
;
         JSR INCRTN         ;INCREMEOT THE RETURN ADDRESS
         LDY #$0
         LDA (RTNADR),Y     ;GET L.O. ADDRESS OF STRING
         STA ZPAGE
         INY
         LDA (RTNADR),Y     ;GET H.O.ADDRESS OF STRING
         STA ZPAGE+$1
;
         JSR INCRTN         ;MOVE RTNADR PAST THE ADDRESS
         JSR INCRTN         ;BYTES
;
;
;  AT THIS POINT, ZPAGE POINTS TO THE STRING WHICH
;  IS SUPPOSED TO BE OUTPUT
;
         DEY                ;RESET Y REG TO ZERO
         LDA (ZPAGE),Y      ;GET THE LENGTH OF THE STRING
         STA LENGTH         ;AND STORE IT IN "LENGTH"
PRTS1    INY                ;MOVE TO THE NEXT CHARACTER
         CPY LENGTH         ;ARE WE THROUGH YET?
         BEQ PRTS2
```

```
;
        LDA (ZPAGE),Y     ;GET THIS CHARACTER
        JSR COUT          ;AND OUTPUT
        JMP PRTS1         ;MOVE TO NEXT CHAR AND REPEAT
;
PRTS2   LDA ASAVE         ;RESTORE THE REGISTERS
        LDY YSAVE
        JMP (RTNADR)      ;SIMULATE AN RTS
;
;
;
ASAVE   EPZ $0            ;ZERO PAGE WORKSPACE
YSAVE   EPZ ASAVE+$1
ZPAGE   EPZ YSAVE+$1
RTNADR  EPZ ZPAGE+$2
;
COUT    EQU $FDED         ;COUT ROUTINE
        END
```

此程式的呼叫是由 JSR　PRTSTR　指令，緊接著輸出字串的位址所組成。

範例：

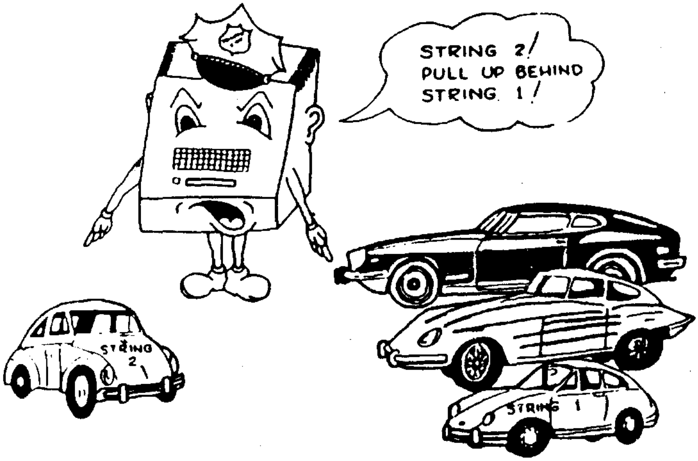
```
        JMP START
STRING  STR "HELLO THERE"
;
START   JSR PRTSTR
        ADR STRING


        ETC.
```

字串串連　(*STRING CONCATENATION*)

字串串連是把二個字串連成一個字串。通常把結果存在第三個字串。

作法如下：首先把二字串的長度相加，若和小於目的地字串的長度時，則沒問題，否則要報告發生錯誤。若沒錯誤，則把和存在目的地字串的第一個位元組，作爲新字串的長



"STRING CONCATENATION"

度。其次把第一個字串移到目的地字串，接著再移第二個字串，就接在第一個字串之後。下面程式把 STR1 和 STR2 串連，結果放在 STR3：

```
; STRING CONCATENATION EXAMPLE
;
;
;
;
; FIRST, CHECK LENGTHS
;
        CLC
        LDA STR1
        ADC STR2
        BCS ERROR        ;> 255 CHARS IS ALWAYS BAD
        STA STR3         ;STORE LENGTH IN STR3
        LDA MAXLEN       ;GET MAXIMUM LENGTH OF STR3
        CMP STR3         ;AND COMPARE TO DESIRED LENGTH
        BLT ERROR        ;IF LESS THAN, AN ERROR MUST
                         ;BE FLAGGED
;
;
; THINGS ARE FINE HERE, SO MOVE STR1 TO STR3
;
```

```
        LDY #$0
CONCT1  LDA STR1+$1,Y     ;GET CHAR FROM STR1
        STA STR3+$1,Y     ;AND MOVE TO STR3
        INY
        CPY STR1          ;DONE YET?
        BLT CONCT1
;
;
; NOW, TRANSFER STR2 TO THE TAIL END OF STR3
;
        LDX #$0
CONCT2  LDA STR2+$1,X     ;GET CHAR FROM STR2
        STA STR3+$1,Y     ;TRANSFER TO STR3
        INY
        INX
        CPX STR2          ;DONE YET?
        BLT CONCT2

        ETC...
```

次字串運算　（*SUBSTRING OPERATIONS*）

次字串運算是很重要的字串函數，可把字串的一部分取出而設定到另一個字串。

要作此函數需要 4 種資料：(1)原始字串的位址。(2)目的字串的位置。(3)次字串的起始位置。(4)次字串的長度。要先檢查所指明的長度是否超過目的字串所允許的長度，若沒有超過，則長度就存在目的字串的第一個位置。另外還要檢查有原始字串中從指明的索引開始是否有夠多的字元（如指明的長度一樣多），若沒有則要報告發生錯誤。下面程式從位置 STR1 上的字串中的 START 位置開始，取出長度爲 LENGTH 的次字串。

```
; SUBSTRING EXAMPLE
;
STR1    EPZ $0
STR2    EPZ STR1+$2
START   EPZ STR2+$2
LEN1    EPZ START+$1
MAXSTR  EPZ LEN1+$1
INDEX   EPZ MAXSTR+$1
LENGTH  EPZ INDEX+$1
;
;
SUBSTR:
        PHP
        PHA
        TYA
        PHA
;
;
; CHECK TO SEE IF LENGTH OF SUBSTRING IS GREATER
; THAN THE LENGTH OF STR2 (PASSED IN MAXSTR)
;
        LDA MAXSTR
        CMP LENGTH
        BLT ERROR

;
; CHECK TO SEE IF ENOUGH CHARS IN STR1
;
        CLC
        LDA INDEX
        BEQ ERROR          ;INDEX OF ZERO NOT ALLOWED
        ADC LENGTH
        BCS ERROR          ;IF > 255 THEN ALWAYS AN ERROR
        LDY #$0
        LDA (STR1),Y       ;GET LENGTH OF SOURCE STRING
        CMP LEN1           ;SEE IF GREATER OR EQUAL
        BLT ERROR          ;ERROR OTHERWISE
;
; NOW, TRANSFER THE SUBSTRING
;
        LDA LENGTH
        STA (STR2),Y       ;INIT LENGTH
        CLC                ;SET UP POINTER TO BEGINNING
        LDA STR1           ;OF SUBSTRING
        ADC INDEX
        STA STR1
        BCC SUBST1
        INC STR1+$1
;
SUBST1  INC STR2           ;INCREMENT PAST LENGTH BYTE
        BNE SUBST2
        INC STR2+$1
;
SUBST2  CPY LENGTH
        BGE SUBST3
        LDA (STR1),Y
        STA (STR2),Y
        INY
        JMP SUBST2
```

```
SUBST3  PLA
        TAY
        PLA
        PLP
        RTS
```

字串的比較（*STRING COMPARISONS*）

最重要的字串函數是字串的比較函數，而字串的關係定義如下：

(1)若二字串長度相等，而且第一個字串的每一個字都等於第二個字串相對位置上的字，則二字串相等。

(2)若二字串長度不等或其中某一位置的字不等，則二字串不等。

(3)從第一個字元開始比較，直到某字串已結束為止，若第一個字串中某位置的字小於第二個字串中該位置的字，則說第一個字串小於第二個字串。若二字串長度不等，但較短字串的字都等於較長字串相對應的字，則說較短字串小於較長字串。例如" ABC "小於" SUN "，而且也小於" ABCD "。這種順序稱為辭典編纂順序（ lexicographical ordering ），為字典所採用的順序法。

(4)某字串大於另一字串的條件和小於的條件一樣，只要把(3)中的所有"小於"都改成"大於"就對。

下面的程式可以測試等於／不等，小於／不小於和大於／不大於等等情況。在每種測試中，若第一種情況為眞（如等於，小於或大於），則累積器的值為"眞"（ $ 1 ），若第二種情況為眞（如不等，不小於或不大於），則累積器為

“假”（$0）。第一個字串的指標在（STR1，STR1+$1）上，而第二個字串的指標則在（STR2，STR2+$1）上。在呼叫此程式之前，需先設定好這個位置值。

```
; STRING COMPARE #1
; TEST FOR EQUALITY
;
; THIS ROUTINE COMPUTES THE COMPARISON
;       (STR1) = (STR2)
; AND RETURNS TRUE OR FALSE IN THE ACCUMULATOR
;
STREQU:
        PHP                 ;PRESERVE C & V FLAGS
        TYA
        PHA                 ;SAVE THE Y REGISTER
;
        LDY #$0
        LDA (STR1),Y
        CMP (STR2),Y        ;COMPARE LENGTHS
        BNE NOTEQL          ;AND QUIT IF NOT EQUAL
;
; IF. LENGTHS ARE EQUAL, SET UP INDICIES
; TO THE BEGINNING OF THE STRINGS
;
        STA LENGTH          ;SAVE LENGTH OF STRINGS
        INC STR1
        BNE SEQU1

        INC STR1+$1
;
SEQU1   INC STR2
        BNE SEQU2
        INC STR2+$1
;
SEQU2   LDA (STR1),Y        ;PERFORM COMPARISONS
        CMP (STR2),Y
        BNE SEQU3
        INY
        CPY LENGTH
        BLT SEQU2
;
;
; THE STRINGS ARE EQUAL HERE
;
;
        JSR DECSTR          ;RESTORE STR1,STR2
;
        PLA                 ;RESTORE Y & PSW REGISTERS
        TAY
        PLP
        LDA #TRUE           ;RETURN TRUE
        RTS
```

```
;
;
; STRINGS ARE NOT EQUAL HERE
;
SEQU3   JSR DECSTR
NOTEQL  PLA                 ;RESTORE Y & PSW
        TAY
        PLP
        LDA #FALSE          ;RETURN FALSE IF NOT EQUAL
        RTS
;
;
;
;
; DECSTR- RESETS STR POINTERS TO THEIR ORIGINAL
;         VALUES
;
DECSTR:
s66     LDA STR1              ;RESTORE STRn POINTERS
        BNE SEQU4
        DEC STR1+$1
SEQU4   DEC STR1
        LDA STR2
        BNE SEQU5
        DEC STR2+$1
SEQU5   DEC STR2
        RTS
;
;
;
;
;
; STRING COMPARE #2
; TEST FOR LESS THAN
;
; THIS ROUTINE COMPUTES
;   STR1 < STR2
;
; ON RETURN, IF STR1 < STR2 THEN THE ACCUMULATOR IS
; RETURNED WITH TRUE.  IF STR1 >= STR2 THEN THE
; ACCUMULATOR IS RETURNED WITH THE VALUE FALSE
;
;
STRLES:
        PHP                 ;PRESERVE C & V FLAGS
        TYA                 ;SAVE Y REGISTER
        PHA
;
        LDY #$0
        LDA (STR2),Y        ;COMPUTE THE MINIMUM LENGTH
        STA MINLEN
        CMP (STR1),Y
        BGE STRLS1
        LDA (STR1),Y
        STA MINLEN
```

```
;
STRLS1  INY              ;TEST LOOP
        LDA (STR1),Y
        CMP (STR2),Y
        BGE NOTLES
        CPY MINLEN
        BLT STRLS1
        BEQ STRLS1
;
; ALL CHARACTERS UP TO THE MINIMUM LENGTH ARE EQUAL
; NOW SEE IF THE LENGTH OF STR1 IS LESS THAN THE
; LENGTH OF STR2
;
        LDY #$0
        LDA (STR1),Y
        CMP (STR2),Y
        BGE NOTLES
;
; NOW STR1 < STR2
;
        PLA              ;RESTORE THE Y REGISTER
        TAY
        PLP              ;RESTORE PSW
        LDA #TRUE        ;TRUE BECAUSE STR1 < STR2
        RTS
;
;
NOTLES  PLA
        TAY
        PLP
        LDA #FALSE
        RTS
;
;
;
; STRING COMPARE #3
; TEST TO SEE IF STR1 > STR2
;
; THIS ROUTINE COMPUTES THE RELATION:
;       STR1 > STR2
;
; THE ACCUMULATOR IS RETURNED WITH THE VALUE TRUE ($1)
; IF THE RELATION HOLDS, FALSE ($0) IS RETURNED
; OTHERWISE.
;
;
STRGTR:
        PHP
        TYA
        PHA
;
        LDY #$0          ;GET THE MINIMUM LENGTH
        LDA (STR2),Y
        STA MINLEN
        CMP (STR1),Y
        BGE SGTR1
        LDA (STR1),Y
        STA MINLEN
```

```
;
SGTR1   INY
        LDA (STR2),Y
        CMP (STR1),Y
        BGE NOTGTR
        CPY MINLEN
        BLT SGTR1
        BEQ SGTR1
;
; STRINGS ARE EQUAL UP TO THE MINIMUM LENGTH
;
        LDY #$0
        LDA (STR1),Y
        CMP MINLEN
        BEQ NOTGTR
;
        PLA
        TAY
        PLP
        LDA #TRUE
        RTS
;
NOTGTR  PLA
        TAY
        PLP
        LDA #FALSE
        RTS
;
;
;
TRUE    EQU $1
FALSE   EQU $0
STR1    EPZ $0
STR2    EPZ STR1+$2
        END
```

同樣地，較好的參數傳送技巧留給讀者作為習題，這裡只是作範例。若在 JSR 之後直接傳送位址會更實際些，就如同前面的輸出程式一樣。而呼叫這程式的方式如下：

```
JSR STREQU          ;IS STR1 = STR2?
ADR STR1
ADR STR2

- OR -

JSR STRLES          ;IS STR1 < STR2?
ADR STR1
ADR STR2
```

```
        - OR -

        JSR STRGTR        ;IS STR1 > STR2?
        ADR STR1
        ADR STR2

        ETC...
```

字元陣列的處理

(*HANDLING ARRAYS OF CHARACTERS*)

有時候比較的字串的長度並不是可變的，因此就不需測試二字串的長度是否相等了，這可節省一些碼。例如 LISA 所用的指令符號都是三個字，所以只要檢查使用者打進來的確實爲三個字，然後再去符號表中尋找即可。下面的程式從輸入緩衝器（ buffer ）中取出 NUMCHR 這個字，然後與從位置 TABLE 開始的列表中的字來作比較：

```
NUMCHR   EQU $3           ;INIT FOR THREE-CHAR LOOK-UP
PTRSAV   EPZ $0           ;POINTER SAVE AREA
TBLADR   EPZ PRTSAV+$1    ;USED TO HOLD TABLE ADDRESS
INPUT    EQU $200         ;GETLN INPUT BUFFER
;
; THIS ROUTINE IS ENTERED WITH THE X-REGISTER POINTING
; TO THE FIRST CHARACTER TO BE COMPARED IN THE INPUT
; BUFFER (PAGE TWO)

; ON RETURN, THE X REGISTER POINTS TO THE FIRST CHAR
; PAST THE ARRAY OF LENGTH "NUMCHR" (DEFINED ABOVE)
;
;
;
LOOKUP:
         PHP
         TYA
         PHA
;
         STX PRTSAV       ;SAVE INDEX TO CHAR ARRAY
         LDA #TABLE       ;SET UP POINTER TO TABLE
         STA TBLADR
         LDA /TABLE
         STA TBLADR+$1
```

```
;
        LDY #$0
LOOP    LDA INPUT,X
        CMP (TBLADR),Y
        BNE NXTENT
        INX
        INY
        CPY #NUMCHR
        BLT LOOP
;
;  GOOD MATCH HERE,  RETURN TRUE
;
        PLA
        TAY
        PLP
        LDA #TRUE
        RTS
;
;
; CURRENT CHARACTER ARRAY DOES NOT MATCH, SET UP INDEX
; TO THE NEXT ELEMENT IN THE TABLE (IF ONE EXISTS)
;
NXTENT:
        CLC
        LDA TBLADR
        ADC #NUMCHR
        STA TBLADR
        BCC NXTE1
        INC TBLADR+$1
;
; RESTORE X REGISTER
;
NXTE1   LDX PTRSAV
        LDY #$0           ;RE-INIT Y REGISTER
;
;
; CHECK TO SEE IF AT END OF TABLE
;
        LDA TBLADR
        CMP #TBLEND
        LDA TBLADR+$1
        SBC /TBLEND
        BLT LOOP
;
; NO MORE ENTRIES, RETURN FALSE AND LEAVE X REGISTER
; POINTING TO THE BEGINNING OF THE TABLE
;
        PLA
        TAY
        PLP
        LDA #FALSE
        RTS
;
;
;
; SAMPLE TABLE, EACH ENTRY MUST CONTAIN "NUMCHR" NUM
; OF CHARACTERS (IN THIS CASE, THREE)
; OF CHARACTERS (IN THIS CASE, THREE)
;
```

```
;
TABLE    ASC "ABC"
         ASC "DEF"
         ASC "GHI"
         ASC "JKL"
         ASC "MNO"
         ASC "PQR"
         ASC "STU"
         ASC "VWX"
         ASC "YZ "
         ASC "ETC"
TBLEND   EQU *
         END
```

注意 TBLEND 被定義爲列表之後第一個可用的位置。

在 6502 微處理機中，列表的處理速度很快，尤其當列表的長度小於 256 個位元組時更快。上面的程式只是一般性檢查列表，任何長度的列表都可以使用。對長度小於 256 位元組的列表而言，每次增加 Y 暫存器，而不增加 16 位元的某位置，則可以省下許多碼和時間。若以 Y 索引位址法來代替間接由 Y 索引位址法，則可節省更多時間。修改成適合於小的列表的程式如下：

```
; TO THE FIRST CHARACTER TO BE COMPARED IN THE INPUT
; BUFFER (PAGE TWO)
; ON RETURN, THE X REGISTER POINTS TO THE FIRST CHAR
; PAST THE ARRAY OF LENGTH "NUMCHR" (DEFINED ABOVE)
;
;
;
LOOKUP:
         PHP
         TXA
         PHA
         TYA
         PHA
;
; TRANSFER INPUT TO BUFFER SAVE AREA
;
```

```
        LDY #$0
LOOP    LDA INPUT,X
        STA BUFFER,Y
        INX
        INY
        CPY #NUMCHR
        BLT LOOP
;
; NOW, COMPARE BUFFER SAVE AREA TO DATA IN TABLE
;
        LDX #$0
        LDY #$0
LOOPO   LDA BUFFER,X
        CMP TABLE,Y
        BNE NXTENT
        INY
        INX
        CPX #NUMCHR
        BLT LOOPO
;
; A MATCH IS FOUND HERE
;
        PLA
        TAY
        PLA
        TAX
        INX
        INX
        INX         ;LEAVE POINTING AT NEXT CHAR
        PLP
        LDA #TRUE
        RTS
;
;
; INCREMENT TO THE NEXT ENTRY (IF IT EXISTS)
;
NXTENT  CPX #$2
        BGE NXT1
        CPX #$1
        BGE NXT2
        INY
NXT2    INY
NXT1    INY
        LDX #$0
        CPX TBLENG
        BLT LOOPO
;
; END OF TABLE HAS BEEN REACHED
;
        PLA
        TAX                 ;LEAVE X REG POINTING TO CHARS
        PLA
        TAY
        PLP
        LDA #FALSE          ;STRING NOT FOUND
        RTS
```

```
;
;
; SAMPLE TABLE, EACH ENTRY MUST CONTAIN "NUMCHR" NUM
; OF CHARACTERS (IN THIS CASE, THREE)
;
;
;
TABLE    ASC "ABC"
         ASC "DEF"
         ASC "GHI"
         ASC "JKL"
         ASC "MNO"
         ASC "PQR"
         ASC "STU"
         ASC "VWX"
         ASC "YZ "
         ASC "ETC"
TBLENG   EQU *-TABLE
         END
```

例中用 TBLENG 指出表的長度，取代原先用來指出表最後位址的指標。同時請記住Y暫存器僅佔8個位元而已。

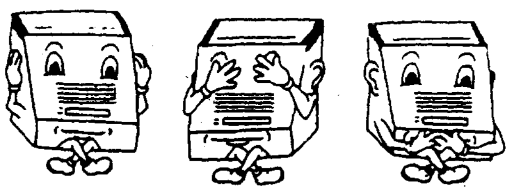
處理文字串的方式不僅限於我們在這章中所提方法，我們僅是提出最受大家喜好的方法介紹給讀者。我們也希望由這幾個例子能使讀者自己能想出其他更好方法來處理文字串的比較問題。

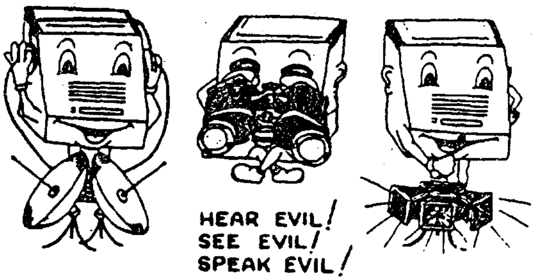
第十五章

APPLE Ⅱ 特殊的輸出入方式

APPLE輸出入結構（*APPLE I/O STRUCTURE*）

APPLE Ⅱ 受到廣大歡迎的原因之一，就是它的輸出入結構與其他機型相比有較好的設計。它有所謂“電動玩具輸出入”連接器配合著類比輸入與數位輸出入綫，雖然這些本來目的是連接電動玩具及一些遊樂儀器用的，但它也可以做爲以印表機連接的界面及RS232綫甚至工業控制用。很多週邊裝置都可以與“電動玩具輸出入”連接而使用，例如光筆就是一個例子。APPLE Ⅱ 的“電動玩具輸出入”可說是它最重要的特色之一，使得它與其他機型相比不但不遜色甚至超過。





" SPECIALIZED I/O "

要了解"電動玩具輸出入"的特色，先從它能提供些什麼樣的輸出入開始。它有三條"旗"的輸入，另加四條"通告者"輸出與四對 8 位元類比轉換成數位輸入裝置。另外還有一條公用激發綫。

除了"電動玩具輸出入"外， APPLE 另外有一些特別的輸出入裝置提供給使用者使用，例如：附在 APPLE II 上面的揚聲器，卡帶輸出入及鍵盤輸入裝置。另外 APPLE II 也有 LORES 與 HIRES 系統來幫助我們繪圖。而控制以上這些裝置與傳送資料給這些裝置時，只需要對APPLE 某些指定記憶體位置做存或取的動作即可。

這些指定的記憶體位置分佈在 $C000 到 $C0FF 的 128 個位元組中。例如先前討論鍵盤輸入時，輸入資料是放在 $C000，如果 $C000 的第七個位元被激發則代表現在 $C000 內存有輸入資料，如果 $C000 第七個位元仍在未激發狀態則代表資料尚未由鍵盤輸入。而 $C000 輸入的資料在接收到之後，我們須將 $C000 第七位元改成未激發狀

態否則等一會又會拿到同樣的字，這項工作由將資料放到$C010 動作完成 。下面就是這個完整的接收鍵盤輸入符號的程式KEYIN：

```
KEYIN   LDA $C000
        BPL KEYIN        ;IF NO KEY PRESSED, LOOP BACK
        STA $C010        ;CLEAR BIT #7 OF THE KEYBOARD
        RTS
```

KEYIN 程式當别的程式須要由鍵盤輸入資料而被呼叫時，它前面的兩個指令就形成一個廻路（ LOOP ）等待輸入資料，直到有了輸入資料之後暫存器A內存著與資料對應的ASCⅡ碼而繼續往第三個指令執行使得資料存入$C010 而 $C000 的第七個位元就恢復成未激發狀態 ，然後回到原先呼叫的程式去繼續執行的工作。

有時候我們只須要知道到底有沒有輸入就可以了，而不須知道輸入資料是什麼。在這種情形下 BIT 指令就能提供我們較方便的檢查方式，我們經由 BIT 指令檢查 $C000 而檢查的結果會造成N旗值的變化，如果有輸入的情形，則N旗在BIT指令執行後會被激發，而 BMI/BPL 指令就可以配合而做轉移（ BRANCH ）的工作而做到如果有輸入則做某一項工作如果沒有資料輸入則做另一項工作的選擇。下列是個檢查是否有輸入資料的"KEYPRS."程式， 它在檢查完之後如果有輸入則在暫存器A存有" TRUE "值（ 1 ），否則就在暫存器A存" FALSE "值（ 0 ）。

```
; FUNCTION KEYPRS.  RETURNS TRUE IF A KEY HAS BEEN
; PRESSED, FALSE OTHERWISE.  THIS VALUE IS RETURNED
; IN THE ACCUMULATOR.
;
;
;
KEYPRS:
        LDA $C000
        ROL               ;SHIFT SIGN BIT (#7) INTO
        ROL               ;THE L.O. BIT OF THE ACC.
        AND #$1           ;MASK OUT ALL BUT BIT #0.
        RTS
```

這個程式首先將資料由$C000 搬到A暫存器中，再經由二次向左方循環的位移（第一次位移到進位位元，第二次位移到最右邊位元）而將第7位元值搬到第0位元位址上。然後與$ 1 做邏輯“及”工作，使得除了原先的第7位元現在的第0位元上的資料外都被剔除了，而第1位元我們前面談到是表示是否有資料輸入。這使得如果有輸入則A暫存器存1否則就存0。

$C020位址是供卡帶輸出時使用的，通常卡帶是提供不具備軟式磁碟系統的 Apple 系統用來儲存數量相當大並且需要保存的資料所用的。但你也可以將這個輸出接到任何擴大系統的輸入而成爲一個聲音的輸出，這與 Apple 本身配備在$C030 的揚聲器的功效就一樣了。正因爲卡帶輸出與 Apple 本身揚聲器可以具有同樣功能，它們的工作方式就十分類似，下面我們就合併討論一下。

一般人類所能聽到的音調頻率大都在20赫茲到2萬 赫茲，也就是每秒 20 個週期到2萬個週期之間。所以理論上你使揚聲器每秒移進移出的做 20 次到2萬次之間，它就會發出“雜音”出來，也就是會產生出一些噪音出來。但因 APPLE II 上所供應的揚聲器僅2英吋大，它不能對理論上的極值做正確的反應，這使得只有在 60赫茲到1萬赫茲之

間的訊號可以由它正常的產生，其他就不保險了。但如果你不用APPLE Ⅱ 本身揚聲器而將卡帶輸出接到一個效果十分好的揚聲擴大系統上，這些問題就不存在了。另外一點，要造成移進移出週期變化你可以簡單的利用一個類似 load 的指令搬動位於$C030 或$C020（卡帶輸出）的資料。而 load 類指令有 LDA, LDX, LDY, BIT, ADC, AND, CMP, CPX, CPY, 與 ORA. 等9個。存回（ Store ）指令則不能夠用來產生搬出搬入的週期變化，這是因爲 6502 寫資料到記憶體內動做的工作方式不能達到要求。因爲6502在發出寫的要求之前都伴隨著有一個讀的動作，兩者相差 92 n 秒，這使得如果你發出寫的指令， 6502先發出一個讀的命令要求揚聲器往某一方向移動，在92n 秒之後又發出一個寫的要求在同一位址造成揚聲器往另外一個方向移動，而一個揚聲器不可能靈敏到這種程度，這兩個動作根本就相互牽制而抵消了，因此移入移出動作根本不會發生而無法導致聲音的產生。

對於公用激發綫的對應位址，APPLE Ⅱ 提供的是$C040，當你由$C040將資料搬到暫存器時（也就是 Load 時）造成公用激發綫在“電動玩具輸出入”連綫器第5個脚上產生一個脈波。如果你將資料由暫存器存囘（ store ）$C040 則在同樣的地方產生二個脈波。但因爲討論這條綫的功能需要涉及一些硬體界面的問題，所以本書中不準備再深入地討論了！

$C050 到$C057 八條綫是供選擇螢光幕顯現方式用的。一般而言，APPLE Ⅱ 提供了兩個記憶體空間供要在螢光幕上顯現的資料儲放用的，一個稱主要（ primary ），另一個稱輔助（ secondary ）。而對於顯現資料的分類也

分成：1）圖形（GRAPHIC），與2）文字（TEXT）兩種，其中圖形資料有兩種系統可以用來顯現圖形：1）LORES與2）HIRES。而也允許圖形與文字混合的表示方式，在此種情形下螢光幕下方有四行空間供文字資料顯現，這些情形的控制都是由$C050到$C057這八條綫來做：

1）$C050：選擇圖形顯現

2）$C051：選擇文字顯現

3）$C052：全部螢光幕都是圖形顯現

4）$C053：螢光幕上半部圖形，下半部四行文字

5）$C054：主要儲存資料顯現選擇

6）$C055：輔助儲存資料顯現選擇

7）$C056：圖形顯現用LORES系統

8）$C057：圖形顯現用HIRES系統

其中要注意圖形狀態下的螢光幕只接收圖形資料的顯現（也就是APPLE Ⅱ本身也處於圖形狀態），相同的文字狀態也有相同限制。

$C058到$C05F八條綫則用來控制四條"通告者"訊號的輸出，這四條"通告者"輸出訊號是TTL 等級的訊號輸出，但如要用以驅動類似LED 類花費電流裝置則須要另外接緩衝器才行。這四條輸出分別是ANφ，AN1，AN2，與AN3。相對應的由$C058 位置開始的一條用以控制的開啓狀態；一條用以控制關閉狀態而成對的8條綫分成四對，$C058與$C059，$C05A與$C05B，$C05C與$C05D，及$C05E與$C05F。對應著$C058與$C059 控制ANφ的開與關，一樣的四對控制綫控制四條"通告者"輸出情形。但是因爲無法知道那四條輸出綫狀態分別是開或關，因此就需要使用者自己記錄了。

$C060 是供卡帶輸入資料使用的 ，卡帶除了速度較磁碟速度慢外，價格則較便宜而且也可以用它將聲音的類比波輸入 APPLE Ⅱ 做數位化的工作。你可以嘗試將高輸出麥克風的輸出連接到卡帶輸入接點而執行下面程式：

```
LOOP     LDA $C060          ;TEST CASSETTE INPUT PORT
         BPL LOOP
         LDA $C030          ;TOGGLE SPEAKER
LOOP2    LDA $C060
         BMI LOOP2
         LDA $C030
         JMP LOOP
         END
```

然後聽一下由揚聲器所發出的聲音。一般對聲音做分析時你就不能像上例中這麼做了，而須要將訊號存到記憶體中，然後再由記憶體存回磁碟中或者卡帶中，如此可使你做採樣的工作而不須考慮額外硬體綫路的配合。

$C061，$C062 ，與 $C063 用以偵測出是否有“旗”的輸入（也就是按鈕是否被按下），它們分別對應著 PB1，PB2 與 PB3 三個按鈕。如果相對應的按鈕被按下則對應的記憶體位址中第 7 個位元就被激發成“ 1 ”否則就恢復成“ 0 ”。下面是一個檢查 PB 按鈕是否被按下的程式，如果有就產生嗶嗶聲，否則就無聲。

```
LOOP     BIT PB1
         BPL LOOP
         LDA #BELL          ;LOAD BELL CHARACTER INTO ACC
         JSR COUT1          ;OUTPUT BELL CHARACTER
         JMP LOOP

PB1      EQU $C061
COUT1    EQU $FDF0
BELL     EQU $87
         END
```

由$C064到$C067 四個記憶體位置是分別對四個“電動玩具輸入”做接收資料用。在起動那四個“電動玩具輸入”之前，你須對$C070 做資料索取的動作，以啓動那輸入裝置中的558時鐘器做類比轉換成數位時記數用。而當類比在轉換成數位過程中，那輸入相對應位址的記憶體中第七個位元將呈現“1”的值，也就是說當相對位址記憶體中值爲負的時候代表類比訊號仍在轉換中。但須注意轉換完值不會出現在相對位址，值的取得是要你自己計算要花多久時間得到正數值。而這在下面例子中有很好的解釋：

```
PREAD    LDA $C070        ;TRIGGER PADDLES
         LDY #$0          ;INIT COUNT
         NOP              ;DELAY REQUIRED FOR HARDWARE
         NOP              ;PURPOSES
;
PREAD2   LDA $C064,X      ;TEST DESIRED PADDLE
         BPL RTS2D
         INY
         BNE PREAD2       ;QUIT IF > $FF
         DEY              ;SET TO $FF
;
RTS2D    RTS
         END
```

其中X內存著那一個“電動玩具輸入”裝置的編號。而在接收完時，值會放在Y暫存器內。而在使用這種輸入時有下列二點須要注意：

i）不可以檢查完一個輸入之後馬上檢查另一個輸入；這是因爲硬體上的一些設計不能允許你如此做，而你可採用下列程式來分隔兩次輸入，以達到延遲的要求。

```
        LDX #$0
LOOP    DEX
        BNE LOOP
```

ii）利用“電動玩具輸入”要比其他輸入浪費許多時間，所以你在用它來處理一些問題時要注意時間的問題。

APPLEⅡ 的輸入不僅提供一些製做電動玩具的輸入工具，對一些儀器的控制所須要的類比訊號的輸入也較其他機器更方便。

第十六章

SWEET-16簡介

SWEET-16

在整數 BASIC 中提供了一個轉換處理器：SWEET-16，它是以編譯器（ interpreter ）方式處理指令的解釋與執行。一般用 6502 提供的機器碼處理 16位元資料須要二個位元組到三個位元組的指令，但SWEET-16 只須要一個位元組長度的指令便可以了，這使得用SWEET-16 寫的程式十分簡潔。本章就在討論SWEET-16的一些特色與優缺點。



"SWEET-16"

首先我們先討論一下什麼是轉換處理器？而編譯器方式又是怎麼回事？所謂轉換處理器，簡單的說便是提供一些原先機器上所沒有的指令，經由這轉換處理器的轉換而執行一些副程式使你能夠執行原先並不存在的指令。也就是一個原先不存在的指令執行時是利用相對應於它的一個副程式來做。而這種處理指令的過程也就是編譯器方式。所以對於 SWEET-16 的每一個指令都有一個副程式來執行，每次 SWEET-16 的一個指令被執行時就造成相對應的程式被呼叫執行。

這種處理指令的方式有什麼缺點？最重要的一個缺點大概就是速度問題了，以 SWEET-16 為例：對同樣的問題採相同的方法去解時，以 6502 的組合語言寫出的程式在速度上較用 SWEET-16 的語言寫出的程式要快 5 到 7 倍。而且對 SWEET-16 而言，它只存在 APPLE II 提供兩種 BASIC 中的整數 BASIC 中而已。如果你不具備整數 BASIC 在你的 APPLE II 系統中時，你有三種方式可供選擇：1）對以下討論跳過，並且忘了有 SWEET-16 的存在，2）借別人有整數 BASIC 的 APPLE II 使用，3）將 SWEET-16 加入到你程式中間。但是其中第 3 條路將花費你不少時間與記憶體空間（1 K 位元組）才能完成。

SWEET-16 到底是怎樣的情形呢？它提供你有 16 個 16 位元暫存器的計算機一樣的一個系統。這些暫存器是用以存於位址及運算中間值用，它們編號由 RO 到 RF（16 進位記數法的 F）。其中 RO，RC，RE 與 RF 是有著特殊功用。RO 視成 SWEET-16 的累積器，在 SWEET-16 系統中只能對 RO 累積器做一些加法、減法、與比較的動作而已。RC 是 SWEET-16 的推疊器指標（STACK POINTER）

，供給副程式呼叫時用。RE 則是存一些運算狀態的暫存器，RF 是程式計數器（PROGRAM COUNTER）。除了以上四個特殊功用的暫存器外的其餘十二個暫存器則都可以供計算操作元位址時用。

與我們在執行十進位運算一樣，用 SWEET-16 系統之前我們須由原先系統跳到 SWEET-16 的系統內，然後才能開始執行你的 SWEET-16 程式，而在執行完你的 SWEET-16 程式之後想要回到原先的系統內也要用一些步驟才能回復原先的系統。到底如何進入 SWEET-16 系統呢？你可用 JSR $F689 指令完成；至於返回原先程式呢？用 RTN 指令即可。下面就是一個呼叫 SWEET-16 系統然後立即又回復到原先系統的程式例子：

```
SW16    EQU $F689
;
        JSR SW16
        RTN
;
        RTS
        END
```

例子中第 2 個指令就是要求進入 SWEET-16 系統的命令，而跟在這指令後面直到 RTN 之前就都應該是 SWEET-16 本身特有指令了，而不能是 6502 的指令。而接著是 SWEET-16 的 RTN 指令這使得進入了 SWEET-16系統之後又再度返回到原先系統內，而在這之後因爲回到原先系統中所以只能用 6502 指令。至於 RTS 則是6502 的指令代表執行完了這副程式而要回到原先呼叫這副程式的主程式或是監督程式中。因此不論你在程式中什麼地方用到 SWEET-16的指令都須使用 JSR $F689 做開頭並用 RTN 指令做結束，否則後果難以預料。

在 SWEET-16 系統中情況暫存器所能記錄的程式狀態有：進位、零值、負值。另外對於負 1 （ $FFFF ）值也有檢查與記錄。

SWEET-16 系統也允許使用者將暫存器做類似 6502 指令中直接搬的動作（ LOAD IMMEDIATED ），也就是將某個 16 位元的值搬到某個暫存器中。這指令的格式如下：

```
SET Rn,<16-BIT VALUE>
```

其中 16 位元的值可以是任何 LISA 所能接受的表示位址方式，而 Rn 中 n 是個 16 進位的數值由 $0 到 $F ，它用來指出你所要把這個 16 位元的值放到 16 個暫存器中那一個。下面就是 SWEET-16 的 SET 指令將值放到暫存器中的例子：

```
LABEL   SET R0,LABEL    ;LOADS THE CURRENT ADDRESS
                        ;INTO R0
        SET R1,$25      ;LOADS $0025 INTO R1
        SET R5,$800     ;LOADS $0800 INTO R5
```

一個 SET 指令佔用 3 個位元組，第一個位元組存 SET 的操作碼，但因資料是 16 個位元所以要用第二個與第三個位元組來存。 SET RF , <VALUE> 是個較特別的情形，因 RF 是SWEET-16 系統的程式計數器，而將程式計數器內容改變成某一個數值也就相當於做跳指令（ JUMP ）的工作一樣。另外對RC與RE兩個暫存器作 SET 指令也須特別注意，因爲這兩個暫存器分別有它的特殊用途，R C 供堆疊器指標用，而R E 則是供狀態暫存器使用。如果你將零數值搬到某個暫存器就會造成狀態暫存器中零指標（ ZERO FLAG ）被激發，否則會恢復成原始狀態，同樣的負 1 （ $FFFF ）被搬到某個暫存器內時也會造成負 1 指標的被激發。但是不管你搬任何數值到任何一個暫存器內都會使進位

指標恢復成原來的狀態。

SWEET-16 提供 LDR 指令使資料可以由某個暫存器搬到累積器內，它的格式為：

```
LDR   Rn
```

其中 n 是某個 SWEET-16 暫存器的編號，由 $0 到 $F。如果零被搬到累積器則會與前面的 SET 指令一樣使零指標被激發，相同情況下負 1 的移動也會造成負 1 指標的被激發，負數也造成負數指標的被激發。而這可以用來檢查累積器，如果你用 LDR R0 則就造成累積器搬回本身的值，但狀態暫存器也因此而依據累積器狀態而激發適當指標。LDR 指令因不須使用 16 位元的資料，因此只佔一個位元而已。

STO（存回暫存器指令）的動作與 LDR 指令相反，它是將累積器內容存回指定暫存器中。它與 6502 指令群中的 TYA 與 TXA 指令的功能類似。與 LDR 一樣會使狀態暫存器中適當的指標被激發，而且指令僅佔一個位元組長度。

但是 SWEET-16 只提供暫存器與累積器資料的搬動而無法做暫存器與暫存器間的資料轉換，這就必須靠二個指令完成，一個是搬到累積器內另一個是由累積器再搬到目地暫存器內。例如要將 R5 內容搬到 R6 內時須要執行下列指令：

```
LDR R5
STO R6
```

而由例子中你可以發現累積器的內容在這個過程中都被破壞了！在 SWEET-16 中的累積器因為常常被使用來傳送資料，因此不適合用來存一些重要的資料，而只能用來存一些計算中所須的操作元而已。

SWEET-16 只提供兩種算術運算，16 位元的加與 16

位元的減。加是用 Sweet-16 的ADD指令執行的，它是將某個指定暫存器的內容加到累積器內而答案自然也就存在累積器中了！ADD 指令的格式如下：

```
ADD Rn
```

n 與前面一樣爲 16 進位的 $ 0 到 $ F 之間任何的一個數字，用來指定累積器與那一個暫存器相加。值得注意的是 $ 0 也被允許，這可以用來將累積器內容乘 2 。相加的結果會造成狀態暫存器內的指標依情形而被激發，例如：第 16 個位元進位到第 17 個位元將造成進位指標的被激發，而因進位不能參予相加動作，這使得進位指標只能供溢位情況檢查用而已。ADD 指令只佔一個位元組的長度。

SUB 指令執行 Sweet-16 系統中相減的工作，它是將指定暫存器的內容由累積器內減去，也就是累積器內容減去暫存器內容，而最後結果與加法一樣放在累積器內。 SUB 指令可用來比較累積器與暫存器內所存的值，這與6502 中的 SBC 指令十分類似。因爲如果累積器內的值大於暫存器內的值則相減的結果會造成進位指標的被激發，而如果小於暫存器內的值則相減則會造成進位指標被回復到原始狀態，相等時會使得零指標被激發，但如不相等則零指標被恢復成原始狀態。所以憑著檢查零指標與進位指標將可以知道暫存器內所存的值與累積器內存的值是那一個較大，但是很可惜的，累積器內的值也隨之被破壞。如果你用 SUB R0 指令將使得累積器被清成零，而SET R 0 , 0 一樣也可以做到；可是用 SUB R0 僅佔用一個位元組而SET R0, 0 須要三個位元組，所以用 SUB R0 可節省程式所佔記憶體的位置。

CPR （比較暫存器）可以用來比較兩個暫存器的內容，它與SUB指令類似，只是相減的結果是放在RD暫存器而不是累積器內，所以累積器內容不會被破壞。

Sweet-16 對於狀態暫存器的檢查與6502 及其他計算機一樣用轉移（ BRANCH ）指令來完成，在 Sweet-16 中對於轉移位址的指定與6502 中一樣採相對位址方式（ RELATIVE ADDRESSING ）。這些轉移有：

BRA ：不管任何狀態都轉移到指定位址。

BNC ：沒有進位指標被激發時轉移到指定的位址。

BIC ：進位指標被激發時轉移。

BIP ：數值檢查爲正數時轉移。

BIM ：數值爲負數時轉移。

BIZ ：零指標被激發時轉移。

BNZ ：數值不爲零或者比較結果不相等時轉移。

BM1 ：數值爲負 1 時轉移。

BNM ：數值不是負 1 時轉移。

BSB ：轉移到 Sweet-16 副程式去。

而所有的轉移指令只佔用兩個位元組。

以上這些轉移指令中， BSB 指令須要我們加以討論一下，以加深各位印象。當 Sweet-16 副程式被 BSB 呼叫時，我們須要將返回的位址資料存到某一個地方才能使我們由副程式返回時回到正確的位址。那麼到底應該存到那裡呢？在前面我們講過 RC 暫存器就是供堆疊器指標使用的，因此當 BSB 指令被執行時，我們返回位址就存放在堆疊器指標所指的位址內，因此我們在執行 BSB 指令之前就須先將 RC 指到某個可用的位址上，不然 RC 有可能隨便亂指而破壞了你的程式或資料，甚至系統。

如何將RC指到某個可用的位址上呢？這可以使用 SET 指令很輕易的達成（這與6502呼叫副程式先用LDX #VALUE 指令設定X暫存器一樣），但是就像前段所提的，位址的選定要十分注意不可以造成任何的錯誤，否則破壞程式或資料事小，破壞了系統那就麻煩了。

要由副程式返回原先呼叫它的程式在 Sweet-16 系統中是用RSB指令而不是6502的RTS指令，更不是用RTN指令的。 RSB 指令只佔用一個位元組的記憶體位置。

暫存器的內容我們可用 INC或DCR 指令做增加 1 或者減 1 的動作。而這兩個指令對暫存器作增或減動作所產生的狀態變化也會在狀態暫存器中反映出來，而供轉移指令作判斷用。 INC 與 DCR 指令都是單位元組長度的型式。

到目前爲止的討論，我們可以發現雖然 Sweet-16 所提供的算術運算與條件檢查類的指令雖然較6502 的要更方便處理 16 位元的資料，但觀念上沒有什麼新的東西。而 Sweet-16 較 6502 好的地方不只限於這些而已，它所提供的指標式指令與資料搬動能力才是它的特色。這使得它能夠使某些動作只須使用一個位元長度的指令，但在6502 中卻須要8 到 16 個位元組的指令才能完成。這些指令大都使用一些暫存器來達成間接位址方式取資料存入累積器內的動作。在下面我們就會一一地討論它們。

首先我們看一下間接方式將資料由記憶體搬到累積器的指令。它的格式如下：

```
LDR@Rn
```

在指令中我們可以看到在操作元 Rn 前面有一個" @ "符號

，它代表與前面看到的 LDR　Rn　採取不一樣的方式，將資料搬到累積器內，這有什麼不同呢？它代表著間接位址方式而且具有下述的特色：這指令本身是將 8 位元的資料由 Rn 指到的記憶體位址搬到累積器中後面的 8 位元上，而累積器前 8 位元資料則改成零，也就是整個累積器的 16 個位元只存由 Rn 所指位址上記憶體的 8 位元資料而已。但在搬完以後，指標 Rn 暫存器還會做自動增 1 的動作而指到記憶體中緊接著剛才位址的下一個位址內，這類指令一般稱為自動增加指令。這種間接方式的 LDR 指令用在記憶體中資料移動與資料尋找的動作中有莫大的方便。我們可以考慮一下下面的例子：

```
START   JSR SW16
        SET R1,$8000
        SET R3,$FF
LOOP    LDR @Rn

        CPR R3          ;CHECK FOR $FF
        BNZ LOOP        ;LOOP IF NOT FOUND
        RTN             ;QUIT SWEET-16, DATA FOUND
                        ;ADDRESS LEFT IN R1
```

這程式是由 $8000 位址開始找 $FF 資料，找到資料 $FF 時，$FF 所在的位址就會存到 R1 暫存器中。

如果你要將連續兩個位元組資料搬到累積器中，你可以用 LDD 指令，它是接連搬的指令。具有下列格式：

```
LDD @Rn
```

它先根據 Rn 所指位址將資料先搬到累積器後面 8 個位元，然後 Rn 自動加 1，接著再利用 Rn 作指標把 Rn 指到位址中的資料搬到累積器的前 8 個位元，最後 Rn 還須要自動加 1 才算整個完成 LDD　@Rn 指令的動作。這也就是把 Rn 指到的接連兩個記憶體上的資料搬到累積器中的方法，而 Rn

也自動加 2 。同樣的狀態暫存器會反映出資料的狀態。

相反的 Sweet-16 也提供了將資料由累積器內經由指標Rn 使用間接位址方式將值存回記憶體中。這指令就是先前所討論的 LDR 指令相反地動作。它具有下列格式：

```
STO @Rn
```

它是根據 Rn 所指到的位址把累積器中後 8 個位元資料存回，然後 Rn 暫存器自動加 1 。它也一樣可改變狀態暫存器內的值，反映出累積器中資料的狀態。這個指令配合著 LDR 指令的使用可以很方便的將一塊記憶體上的資料搬到另外一個地方去。下面就是將 $8000 位址到 $9000 位址上資料搬到 $30000 到 $4000 位址上去的程式：

```
START   JSR SW16
MOVE    SET R1,$8000     ;SET UP POINTER REG #1
        SET R2,$9000     ;SET UP FINAL VALUE REG
        SET R3,$3000     ;SET UP POINTER REG #2
;
LOOP    LDR @R1          ;GET DATA @R1

        STO @R3          ;STORE @R3

        LDR R1
        CPR R2           ;DONE YET?
        BNC LOOP         ;IF NO CARRY (I.E. LESS THAN)
        BIZ LOOP         ;IF EQUAM
        RTN
        BRK
        END
```

如果你希望將累積器 8 個位元與後 8 個位元資料都存到記憶體中連續兩個位址上時，你必須要用 STD 指令，它是做與 LDD 相反的動作。它首先將後 8 個位元上的資料存到 Rn 所指的位址上去，然後 Rn自動加 1 再根據 Rn 暫存器內新的位址將資料由累積器前 8 個位元中存回，最後 Rn 還要

再加1。

最後我們討論一下 Sweet-16 的三個較特殊的指令：POP, STP 與 PPD。POP 指令的格式如下：

```
POP @Rn
```

它是"先"將Rn減1然後根據Rn 所指的位址將資料搬到累積器的後8個位元內。POP 指令可以幫助我們做堆疊器時使用，另外還能用來寫"往右移"的程式。

STP 指令則與 POP 指令所做的工作相反。它是先將 Rn 減1之後再根據 Rn 所指的位址將累積器中後8個位元資料存回。

至於 PPD則相當於 POP 指令，只是重複做二次 POP 動作於不同的累積器部分。它首先將 Rn 減1，然後根據 Rn 將資料由記憶體搬到累積器前8個位元內，再把Rn減1之後同樣的把資料依據 Rn 所指的位址搬到累積器的後8個位元，它具有下列的規格：

```
PPD @Rn
```

雙位元組的堆疊器可用 PPD與STD 指令來處理資料存回與取得的動作。POP , STP與 PPD 指令均是單位元組的指令，而且當這些指令執行完後，進位指標就會被恢復成原始狀態。其中 POP 指令不管減1動作如何做（做多少次）至少都會使Rn內值爲正值，PPD與 STP 動作完成後累積器內的值對狀態暫存器產生影響（激發適當狀態位元）。

SWEET-16硬體的要求

(*SWEET-16 HARDWARE REQUIREMENTS*)

所有 Sweet-16 的暫存器都是利用零頁記憶體空間來模擬的，其中R0對應著$0與$1位址，R1對應著$2與$3 如此一個暫存器用二個記憶體位址來模擬，所以零頁記憶體前32個位元組（由$0到$31）都有著特殊用途，我們在寫程式時要注意不可誤用這些位置而造成錯誤，這 32 個位址也可以供一般系統與 Sweet-16 系統傳送資料用。所有6502暫存器在進入 Sweet-16 時都會被保留存在某個特定的地方，而等到由 Sweet-16 回來時就會自動存回而恢復成原先狀態。最後一點要注意的是由一般系統進入 Sweet-16 系統前的狀態須處於二進位運算狀態否則會產生一些令你感到奇怪的現象。

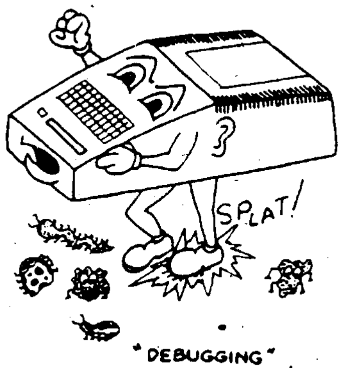
第十七章

程式的偵錯與除錯

一般來講，除非是你寫的程式十分簡單，否則通常都須要更正錯誤而達到你當初設計程式的要求。 LISA 是一個交談式組合語言翻譯器（ interactive assembler ），它可以幫助我們在發現程式中文句結構錯誤（ syntax errors ）時提供很大的幫助，而節省許多時間。但是它並非任何文句結構錯誤都能找出，仍有一些較簡單的錯誤無法偵測出，須要我們另外花少許時間找出來更正。眞正麻煩的錯誤問題是出在程式中隱藏著邏輯錯誤（ logical errors ）。一般常見的邏輯錯誤有：在執行完十進位運算後忘記將十進位運算旗標恢復，或者是將資料當成程式執行，甚至在進入一個副程式之初忘了將暫存器內容存在某一個記憶體區域。不像 BASIC 語言當程式有錯時，在某些情形下執行會停止下來，並且印出一些警告的消息。 6502 組合語言與一般組合語言一樣，照常往下執行，一直到不能執行爲止或遇到你程式中要求停止執行的指令爲止，而這時因你程式有錯可能造成一些錯得十分離譜的答案或者將一些記憶體中的資料與指令給摧毀了甚至連磁碟機內存的資料也遭不幸。

所幸 APPLE II 微算機提供了幫助我們在執行程式中間

找錯並且更正的工具。其中最重要的是 Apple 的監督程式。它配合著佔 2 K記憶體的程式可以做到程式模擬的工作，將暫存器與記憶體內的資料顯現出來並修正錯誤，同時也可以搬動記憶體內的資料並做檢查的工作。有了它你可以及時修正錯誤的地方後再繼續檢查程式。除了它之外，另外 LISA 也另提供一些供連線（ On-Line ）系統用的偵錯程式。



GO—指令(G)（ *GO COMMAND (G)* ）

GO 指令除了提供我們在執行程式時須要的起動命令外，也可以供我們偵錯時使用。這是因爲可將我們的程式視爲監督程式的一個副程式，而 GO 指令就相當於 JSR 指令。等到程式執行到 RTS 指令時就會由我們的程式返回監督程式內。也就是監督程式是主程式，而由 GO 指令聲明開始執行的程式是監督程式的副程式。例如 800 G 代表由 800 位置開始執行，而 800 開始的程式就是一個監督程式的副程式，

而在這副程式被執行中遇到 RTS 就會使得執行回到監督程式中。

這對我們偵錯到底有什麼用處呢？我們可以將我們所有的程式分成一個個副程式，而分別要求每個被執行，然後檢查執行的結果，看一下是那一個出了問題，然後對於出錯的程式加以仔細檢查。但是這種偵錯方式只能用在副程式是獨立的，不須接受別的副程式傳送資料時才能工作。例如你有位在 $980 , $AC0 與 $1000 三個地方的副程式，當你由你的主程式（監督程式的副程式）開始執行時，發現產生錯誤。這時你就分別利用 900G, AC0G 與 1000G 三次命令分別執行你的三個副程式，再一個個地找出毛病。如果你的程式須要資料相互傳輸則……

暫存器與記憶體的起始值
（*INITIALIZING REGISTERS AND MEMORY*）

除了最簡單的副程式之外，一般副程式都須要一些資料的傳輸，由別的副程式或者主程式得到執行所須的資料。一般來說，傳輸小筆的資料都是利用暫存器，如果大筆資料就利用記憶體內某一塊位址，然後將資料起始位址利用堆疊器（ STACK ）或暫存器或某指定位址傳輸，而達到傳送資料的目的。而在利用 GO 指令分別執行每個副程式而偵錯前我們最好先要知道資料傳輸的內容，或者將資料改成正確的型式，以便能夠確定傳入的資料是否正確，而能判斷不是資料有問題而是程式有問題。

利用 control-E 指令我們可以使得監督程式將全部 6502 暫存器的內容顯現出來。因此在發出 GO 指令之前我們可以

可以先打入 control-E 再打 return 而在螢光幕上得到類似下例的結果：

```
A=0A X=FF Y=D8 P=B0 S=G8
```

供我們先檢查傳入副程式的值是多少。但如果我們發現資料不對而須更正時怎麼辦呢？我們可以在資料顯現之後打一個引號：接著輸入新的資料，如此便可以修正暫存器內的資料了。下面就是一個修正A暫存器（累積器）內容的例子：

範例：

```
*control-E

 A=0A X=FF Y=D8 P=B0 S=F8

*:C1

*control-E

 A=C1 X=FF Y=D8 P=B0 S=F8
```

如果你不想修正A暫存器內容，而只想修正S暫存器內容時則怎麼辦呢？甚至只想修正其中一部分暫存器的內容而不改變其他暫存器時又該怎麼辦呢？其實第一個問題就是第二個問題的一個例子，我們舉這個例子來解釋如何做這種修正的工作。我們只須將每個暫存器希望的值按螢光幕次序排好而每個暫存器值分開輸入並且一個空白來分開。對於不更正的當然就是原先值，而要改變的便是新值了！下面就是將暫存器值更正成FF的例子：

範例：

```
*control-E

 A=C1 X=FF Y=D8 P=B0 S=F8

*:C1 FF D8 B0 FF

*control-E

 A=C1 X=FF Y=D8 P=B0 S=FF
```

下面是一個利用上面所討論的方法檢查一個位在 $FDF0 開始的將存在A暫存器內資料顯現於螢光幕上程式的例子：（其中C1是字符A所對應的碼，C2是字符B對應的碼）

```
*control-E

 A=0A X=FF Y=D8 P=B0 S=F8

*:C1

*FDF0G
A
*control-E

 A=C1 X=FF Y=D8 P=B0 S=F8

*:C2

*FDF0G
B
*control-E     etc...
```

如果你的程式由記憶體傳輸資料，則如何將資料放到記憶體中呢？這與先前的修正暫存器內容的方式有一點點不同。這裡你只須先按位址接著是冒號，再接著便是一個個資料

，資料間仍用空白分隔。下面就是個例子：

範例：

```
*F0:00 80 C0

*800G
```

它假定你程式所須的資料放在 $F0 ，$F1 與 $F2 內而經由它將資料 00,80 與 C0 分別放到上述三位址內。然後你便可用 G O 指令執行你的副程式，看一下程式是否能用這些資料產生你預料中的正確答案，判定程式是否正確。

以上的步驟不一定只用在安排資料方面也可以用來輸入一些試驗程式的機器碼（ machine code ）。例如你想檢查位於 900 開始的 PRINT 程式是否正常，而將下列程式（用以呼叫 PRINT 將 ABC 在螢光幕上打出 ，藉以檢查 PRINT 程式是否正常工作）。

```
JSR $900
ASC "ABC"
HEX 00
RTS
```

所對應的機器碼輸入 1000位址，然後執行它檢查結果。

```
*1000:20 00 09 C1 C2 C3 00 60

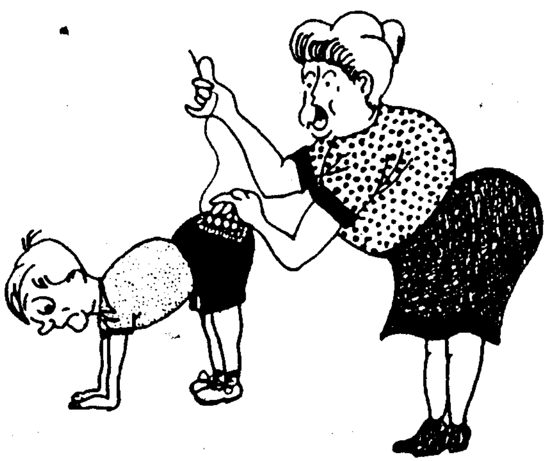
*1000G
ABC
*
```

修正指令碼(補正)
(MODIFYING INSTRUCTION CODE (PATCHING))

在 LISA 被使用以前，修正一個程式地工作通常都十分繁瑣，而使得大部分的程式員只能根據正確的碼修正某個錯誤指令，但如此做程式員須要具備一些額外的知識才能很正確地做更正的工作不至於愈改愈錯。LISA 是方便了許多但仍然稍微嫌麻煩了些，因爲修正完一部分就須重新再翻譯一次，而對有些情形這是沒有必要的。

某些情形下直接地更正機器碼仍是最快的，這些情形有：將相同隱藏式(implied)位址方式的指令做變更(如將 DEX 改成DEY)，要將某段指令除去(將指令改成NOP指令)，或要加上 BRK 指令幫助偵錯。其中後面兩種情形是最爲常用的。

NOP 指令的作用有兩種：1)時間延遲用(2)取代程式中某些不想用的指令。NOP的機器碼爲＄EA，你最好能記



住它。而更正記憶體中指令成 NOP 最快捷地辦法就是利用前面所講的記憶體修正指令（ < addr> : < data> ）來更正某個位址上的指令。例如現在你希望將 $ 890 位址上的指令 PLA 除去則可以使用兩個方法：
(1)利用 LISA 更正原始的程式然後再重新翻譯得到新的更正過的機器碼，但如此做差不多要花費你一分鐘的時間才能做完，但如果你知道所要除去的 PLA 指令所在的位址（ $890 ）就可以使用另一種較快的方式；(2)利用 809 : EA 的修正指令直接輸入 PLA 指令修正成 NOP 指令而達到除去 $890 位址上 PLA 指令的目的，就可以立即重新執行新的更正程式。但是別忘了等下原始程式相對地也須要修正一下，而且如果須要除去的指令有一個到兩個位元組（ byte ）的操作元位址部分也須一併改成 NOP ，不能只改機器碼部分。

在對不是單獨的完整副程式做偵錯時，我們通常用設定停止點方式使得執行到那指令時，會回到監督程式而暫停，使我們可以檢查一些重要的資料，看看內容是否合乎我們的要求。而這個用以設定停止點的指令是 BRK 指令，我們只須將程式中希望停止位置的指令改成 BRK 指令便可以了。但因：(1)正常指令被 BRK 指令取代，(2)堆疊器已被破壞而無法用繼續由設定停止點往下執行，是使用這個指令的缺點。而且不是所有的 APPLE II 微算機都可以執行 BRK 指令。

在 Lazer 系統中有個 TRACE / 65 的程式可以幫助我們檢查程式的錯誤、更正錯誤。它是個交談式的偵錯工具，它提供 APPLE II 一個類似 CP/M 系統中偵錯程式（ DEBUGGER ）的功能。它允許使用者控制著程式，一個指令一個指令的執行，並且在執行完一個指令之後能夠檢查所有暫存器的內容。它也允許使用者設定一些沒有破壞性的停止點而允許使用者有能力在螢光幕上將機器碼相對的指令符號顯

示出來。一般而言 TRACE/655 可以減少我們一半的偵錯時間。

要用 TRACE/65 則你必須用 LISA 翻譯你放在$800與$47FF 區域中間的程式。一旦翻譯完成，而機器碼也在指定位址放好之後，螢光幕上會出現一個"）"符號，你接著按下一個T字再按RETURN 鍵。這時 TRACE/65 就會向你要求程式的起始位址；等你輸入這資料，它就會根據這位址開始執行；在執行的時候， TRACE/65 會將指令打出在螢光幕上，如果你想要停止執行下個指令你可以按 space 鍵暫停執行。如要繼續就再按 space 即可。

TRACE/65 的操作可分成兩種狀態：(1)執行狀態與(2)參數狀態。在執行狀態時，程式被執行而且被顯現於螢光幕上。而在參數狀態你可以設定停止點，修正記憶體內資料與暫存器內容，並且可以由參數狀態返回到執行狀態繼續執行程式。

要進入參數狀態，我們可以按一下空白鍵然後輸入一個P字，即可使執行停止下來然後進入到參數狀態。這個時候螢光幕會出現所有參數狀態下可以使用的指令與它們的簡單解釋，這可幫助你使用參數狀態的指令。如果你選擇按"A"鍵，則就是選擇將你程式中被執行的指令顯現在螢光幕上，供你決定或參考用。如果你不選擇這種顯現的狀態則程式進行的步驟你將無法由螢光幕得知，但如此執行的速度較顯現狀態下執行的速度要快上100倍。因此在不須要檢查的部分你可以用下面討論的停止點設定方式要它自己獨立執行，而到停止點時再改成顯現狀態而將內容顯現以確定問題的癥結所在，如此可節省不必要的時間浪費。

除了"A"之外，參數狀態尚有"B"的選擇提供給你

，“ B ”狀態是用以設定一些停止點，也就是設定一些位址使計算機執行到這些位址時能夠暫停下來，這時使用者便可檢查一下資料是否正確，以確定程式是否有錯，錯在那裡。與先前討論的 BRK 指令不同，它可在檢查完之後由停止點再往下執行程式。當你按下“ B ”鍵則 TRACE/65 會向你要停止點的所在位址，而一次最多可決定四個停止點給 TRACE/65 系統，等下執行時在遇到這些你設定的停止點時便會停止執行等待你下一步的要求。

“ C ”鍵功用在於一旦你不小心誤按了“ P ”鍵而進入參數狀態，但你不想做任何動就離開這種狀態時使用。它會使你的回復到原先誤按“ P ”鍵前的狀態。

“ D ”提供你在檢查完你希望檢查的程式部分後，想停止程式的執行，離開偵錯狀態時使用。

先前所討論的都是檢查程式在執行中的情形，但如何在發現錯誤時修正它，而讓程式能夠依照正確的方式往下執行，以便繼續檢查程式下面的部份是否有錯，而不須重新修正程式，翻譯修正後程式然後再執行。這樣做太費時間了！ TRACE/65 提供了“ $ ”指令，讓我們在參數狀態下能夠修正資料內容而不須做上述的步驟。而用“ $ ”修正記憶體中指令或者資料甚至暫存器內的資料可以用‘ $＜ 10C＞：＜ data＞’方式，要修正暫存器的內容則可利用修正零頁記憶中 $DA 到 $E9 位址的內容而達到。下列是這些位址與暫存器對應的關係：

```
PC   :$DA, $DB
ACC  :$E5
XREG:$E6
YREG:$E7
PSW  :$E8
SP   :$E9
```

除了上述的 $DA 到 $E9 位址外，零頁記憶體內容在你使用 TRACE/65 系統偵錯時只准許你修正 $D6 到 $D9 與 $EA 到 $FF 的位置，也就是整個零頁記憶體中只准許你修正 $D6 到 $FF 的內容，其他一概不允許，否則會有令你無法理解的錯誤出現。

程式偵錯例子（*PROGRAM DEBUGGING SESSION*）

我們看下面所要討論的例子就是下列的程式：

```
START:
        LDX     #$0
LOOP    LDA     MSG,X
        BEQ     QUIT
        JSR     $FDED

        DEX
        BNE     LOOP
;
MSG     ASC     "THIS IS A TEST"
        HEX     00
;
QUIT    BRK
        END
```

這程式中有個非常重大的錯誤，那就是在利用索引位址方式（ INDEX ADDRESSING ）拿資料放到暫存器A時，索引暫存器（ INDEX REGISTER ）的內容應是遞增的，也就是第 5 個指令 DEX 應是 INX 。否則就會有一大堆廢物被印出來，而與當初要求印出“ THIS IS A TEST ”不同。那你應該如何使用 TRACE/65 來找那個錯誤呢？首先你須用 DOS 指令中的“ BLOAD PROGRAM . ”將你的程式放到記憶體中，然後在執行前先設定停止點，否則等下程

式便會從頭執行到尾，如此根本無法幫你找出錯誤。停止點要設定在那裡呢？當然對於監督程式中COUNT程式不須檢查；依照我們先前討論的一個個副程式分別檢查的原則，我們設定在 $FDED ，也就是將暫存器A的內容印出的副程式的第一個指令位址，如此可使我們在進入執行這個副程式之前先確定資料輸入是否正確。所以我們先按“ P ”鍵進入參數狀態，然後按“ B ”鍵要求輸入停止點。由於輸入停止點的數目可以有四個，所以我們必須指出現在輸入的第幾個。假設是第一個則按“ 1 ”然後 TRACE/65 會向你要求位址，這時你就可以輸入 FDED，然後便可以開始執行你的程式了。

要 TRACE/65 開始以偵錯狀態執行你程式，你必須按下“ T ”鍵，然後TRACE/65 會向你要開始執行的位址，一旦你輸入之後，執行便開始。由於你先前設定FDED是停止點所以在執行到FDED位址內的指令前TRACE/65 會停止執行，然後打出下列消息讓你確定現在程式是停在那一個停止點，‘ BREAK POINT ENCOUNTERED AT LOCATION $nnnn ’並且暫存器內容也會展現在螢光幕上，你可以發現暫存器A的內容為‘ T ’的碼（ $DA ），這就確定到目前為止程式執行的十分正常。下面我們不希望位於 $FDED 的監督程式中COUNT程式被執行（因為它既然是系統一部分則必定經過廠商偵錯通過，所以不須要考慮錯出在COUNT中），因此我們利用按“ P ”鍵進入參數狀態，然後利用 $DA:58FF 修正程式計數器（ PC ）內容讓它指到COUNT中RTS 指令，回到我們的程式部分。但如何回來呢？只要在修正程式計數器（ P C ）之後按“ C ”鍵讓程式再繼續執行，這時它經由第6個指令回到程式再執行

程式一次，然而這次在停止點停止時暫存器A內容就不是你所希望的了！如此做幾次下來相信你便可找出 DEX 指令的錯誤而更正。

這例子顯示了我們在偵錯時，可以使用停止點方式跳過不會有錯的程式而直接到有疑問部分的起始地方停止下來，慢慢的檢查程式這個有疑問的部分。

附錄A

APPLE Ⅱ 的表格圖

導　論

在本附錄中我們列舉出使用APPLEⅡ 計算機所要用到的一些表格、圖。利用這些可幫助讀者您在使用APPLE Ⅱ時，能方便且有效率。

表二　鍵及其ASCII 碼

Key	Alone	CTRL	SHIFT	Both	Key	Alone	CTRL	SHIFT	Both
space	$A0	$A0	$A0	$A0	RETURN	$8D	$8D	$8D	$8D
0	$B0	$B0	$B0	$B0	G	$C7	$87	$C7	$87
1!	$B1	$B1	$A1	$A1	H	$C8	$88	$C8	$88
2"	$B2	$B2	$A2	$A2	I	$C9	$89	$C9	$89
3#	$B3	$B3	$A3	$A3	J	$CA	$8A	$CA	$8A
4$	$B4	$B4	$A4	$A4	K	$CB	$8B	$CB	$8B
5%	$B5	$B5	$A5	$A5	L	$CC	$8C	$CC	$8C
6&	$B6	$B6	$A6	$A6	M	$CD	$8D	$DD	$9D
7'	$B7	$B7	$A7	$A7	N^	$CE	$8E	$DE	$9E
8(	$B8	$B8	$A8	$A8	O	$CF	$8F	$CF	$8F
9)	$B9	$B9	$A9	$A9	P@	$D0	$90	$C0	$80
:*	$BA	$BA	$AA	$AA	Q	$D1	$91	$D1	$91
;+	$BB	$BB	$AB	$AB	R	$D2	$92	$D2	$92
,<	$AC	$AC	$BC	$BC	S	$D3	$93	$D3	$93
-=	$AD	$AD	$BD	$BD	T	$D4	$94	$D4	$94
.>	$AE	$AE	$BE	$BE	U	$D5	$95	$D5	$95
/?	$AF	$AF	$BF	$BF	V	$D6	$96	$D6	$96
A	$C1	$81	$C1	$81	W	$D7	$97	$D7	$97
B	$C2	$82	$C2	$82	X	$D8	$98	$D8	$98
C	$C3	$83	$C3	$83	Y	$D9	$99	$D9	$99
D	$C4	$84	$C4	$84	Z	$DA	$9A	$DA	$9A
E	$C5	$85	$C5	$85	←	$88	$88	$88	$88
F	$C6	$86	$C6	$86	→	$95	$95	$95	$95
					ESC	$9B	$9B	$9B	$9B

所有之值均爲十六進制，十進制值可從表三獲得。

表三：ASCII 字元組

十進制：		128	144	16Ø	176	192	2Ø8	224	24Ø
十六進制：		$8Ø	$9Ø	$AØ	$BØ	$CØ	$DØ	$EØ	$FØ
Ø	$Ø	nul	dle		Ø	@	P		p
1	$1	soh	dc1	!	1	A	Q	a	q
2	$2	stx	dc2	"	2	B	R	b	r
3	$3	etx	dc3	#	3	C	S	c	s
4	$4	eot	dc4	$	4	D	T	d	t
5	$5	enq	nak	%	5	E	U	e	u
6	$6	ack	syn	&	6	F	V	f	v
7	$7	bel	etb	'	7	G	W	g	w
8	$8	bs	can	(	8	H	X	h	x
9	$9	ht	em	)	9	I	Y	i	y
1Ø	$A	lf	sub	*	:	J	Z	j	z
11	$B	vt	esc	+	;	K	[	k	{
12	$C	ff	fs	,	<	L	\	l	|
13	$D	cr	gs	—	=	M	]	m	}
14	$E	so	rs	.	>	N	ˇ	n	-
15	$F	si	us	/	?	O	_	o	rub

表中兩個或三個小寫字母的組合是標準ASCII控制字元的縮寫。並不是表中所列的所有字元均可由APPLE鍵盤產生，尤其是表中最左兩縱行（用小寫字母組合表示）的字元，如符號〔（左括弧），（後斜線），＿（底線）及控制字元"fs"，"us"及"rub"。

表中任何一個字元的十進或十六進制值是將該字元所對應位置之上方及左方十進值或十六進值相加的結果。

APPLE視頻顯示器

APPLE視頻顯示器

顯示方式：記憶器影射入系統RAM

顯示形式：正文，低解析度繪圖，
高解析度繪圖

正文容量：960字元（24横列，40縱行）

字元方式：5 × 7矩陣

字元組：大寫ASCII，共64個字元

字元形式：正常，相反，閃爍

繪圖能力：1920方塊（低解析度）
即分爲40 × 48陣列
53760點（高解析度）
即分爲280 × 192陣列

顏色數：16（低解析度）
6（高解析度）

圖1顯示了APPLE之顯示幕在正文模式下記憶器之分配圖，即顯示幕上每個字元的相對應記憶器位址。

表8　低解析度繪圖色彩

十進制	十六進制	色彩	十進制	十六進制	色彩
Ø	$Ø	黑色	8	$8	棕色
1	$1	紫紅色	9	$9	橙色
2	$2	深藍色	1Ø	$A	灰2
3	$3	紫色	11	$B	粉紅色
4	$4	深綠色	12	$C	淺綠色
5	$5	灰1	13	$D	黃色
6	$6	中藍色	14	$E	碧綠色
7	$7	淺藍色	15	$F	白灰

表7　ASCII顯示幕字元

		反	相			閃	爍			(控	制)	正	常			(小寫)	
十進制		∅	16	32	48	64	8∅	96	112	128	144	16∅	176	192	2∅8	224	24∅
	十六進制	$∅∅	$1∅	$2∅	$3∅	$4∅	$5∅	$6∅	$7∅	$8∅	$9∅	$A∅	$B∅	$C∅	$D∅	$E∅	$F∅
∅	$∅	@	P		∅	@	P		∅	@	P		∅	@	P		∅
1	$1	A	Q	!	1	A	Q	!	1	A	Q	!	1	A	Q	!	1
2	$2	B	R	"	2	B	R	"	2	B	R	"	2	B	R	"	2
3	$3	C	S	#	3	C	S	#	3	C	S	#	3	C	S	#	3
4	$4	D	T	$	4	D	T	$	4	D	T	$	4	D	T	$	4
5	$5	E	U	%	5	E	U	%	5	E	U	%	5	E	U	%	5
6	$6	F	V	&	6	F	V	&	6	F	V	&	6	F	V	&	6
7	$7	G	W	'	7	G	W	'	7	G	W	'	7	G	W	'	7
8	$8	H	X	(	8	H	X	(	8	H	X	(	8	H	X	(	8
9	$9	I	Y	)	9	I	Y	)	9	I	Y	)	9	I	Y	)	9
1∅	$A	J	Z	*	:	J	Z	*	:	J	Z	*	:	J	Z	*	:
11	$B	K	[	+	;	K	[	+	;	K	[	+	;	K	[	+	;
12	$C	L	\	,	<	L	\	,	<	L	\	,	<	L	\	,	<
13	$D	M	]	–	=	M	]	–	=	M	]	–	=	M	]	–	=
14	$E	N	ˇ	.	>	N	ˇ	.	>	N	ˇ	.	>	N	ˇ	.	>
15	$F	O	_	/	?	O	_	/	?	O	_	/	?	O	_	/	?

表7　ASCII顯示幕字元組

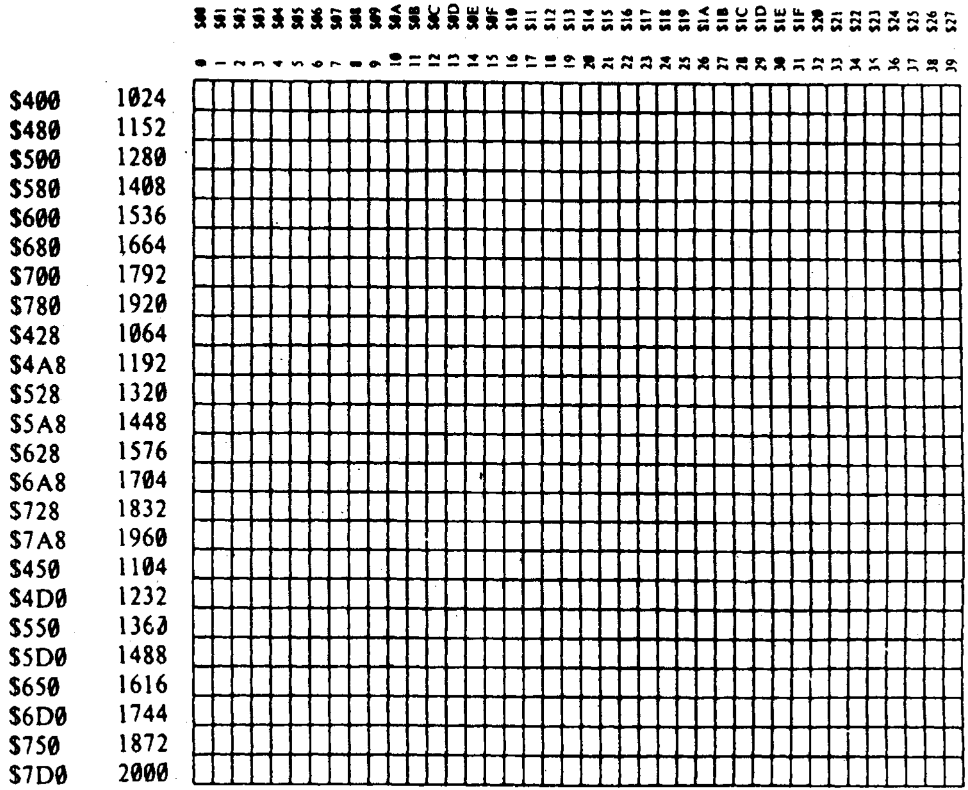

$00 $01 $02 $03 $04 $05 $06 $07 $08 $09 $0A $0B $0C $0D $0E $0F $10 $11 $12 $13 $14 $15 $16 $17 $18 $19 $1A $1B $1C $1D $1E $1F $20 $21 $22 $23 $24 $25 $26 $27
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
$400 1024
$480 1152
$500 1280
$580 1408
$600 1536
$680 1664
$700 1792
$780 1920
$428 1064
$4A8 1192
$528 1320
$5A8 1448
$628 1576
$6A8 1704
$728 1832
$7A8 1960
$450 1104
$4D0 1232
$550 1360
$5D0 1488
$650 1616
$6D0 1744
$750 1872
$7D0 2000

圖 1　正文顯示幕分配圖

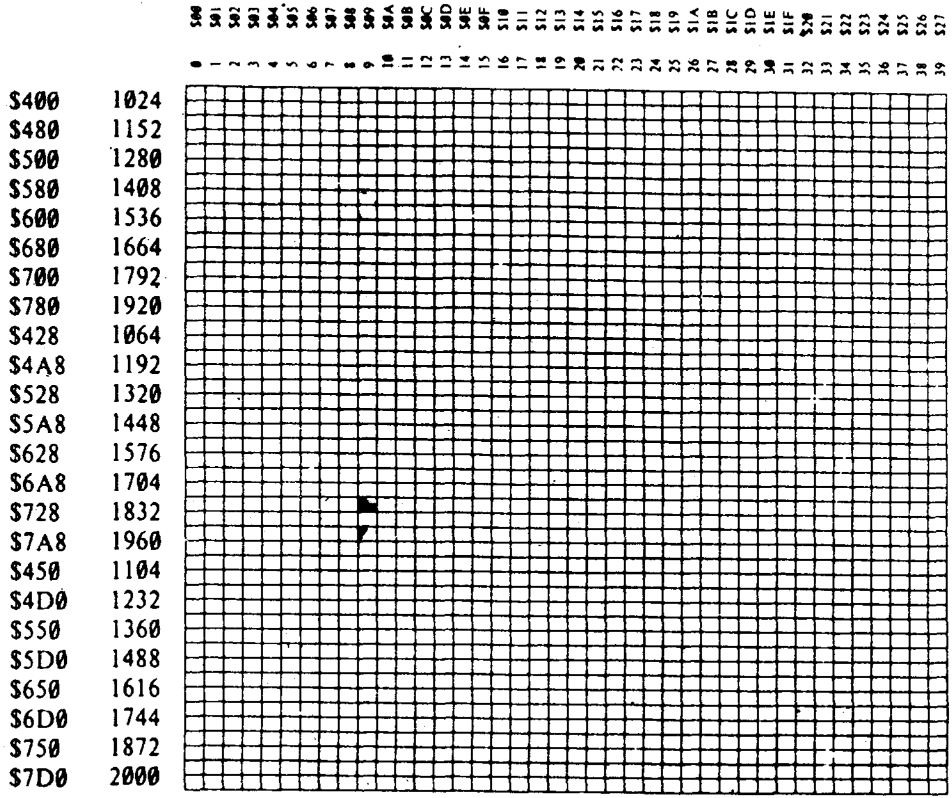


圖 2　低解析度繪圖模式記憶器分配圖

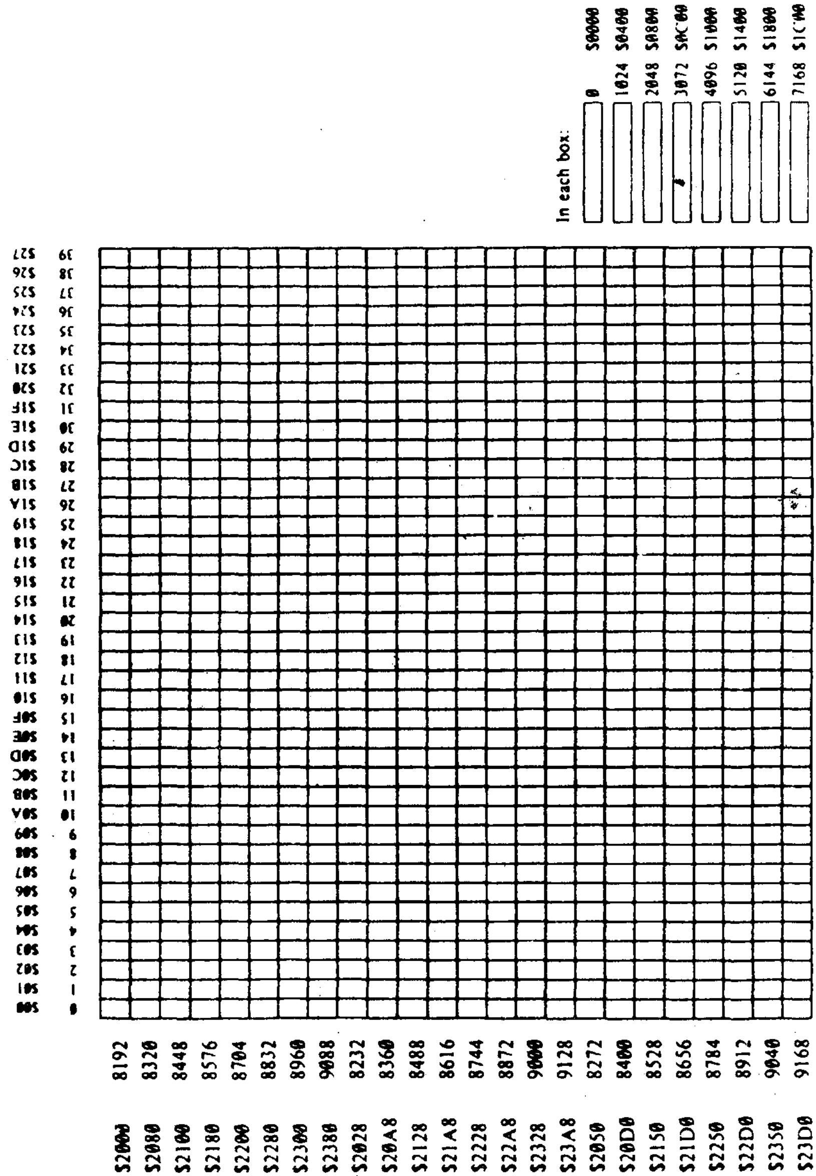


圖 3 高解析度顯示幕分配圖

在每方塊中欲得任何位元組的位置，將該位元組之橫列方塊，縱行方塊及在該方塊之位置相加。

監督程式命令總結

監督程式命令總結

檢視記憶器內含	
{位址}	檢視一個位置之內含。
{位址 1}.{位址 2}	顯示所有位於{位址 1}及{位址 2}之間的所有內含。
RETURN	顯示最後開啓位置之後的內含，最多 8 個位置。
改變記憶器之內含	
{位址}:{數值}{數值}……	將數值存入以{位址}爲開始的連續位置內。
:{數值}{數值}……	將數值存入以下次可改變位置爲開始的連續位置內。
移動與比較	
{目的地}<{啓始}.{最終}M	將{啓始}.{最終}範圍內的資料抄至以{目的地}爲開始的範圍。
{目的地}<{啓始}.{最終}V	比較{啓始}.{最終}與以{目的地}爲最始之記憶器內含。
卡帶之資料儲存	

{啓始}.{最終}W	將位於{啓始}.{最終}範圍內之記憶器內含儲存在卡帶上，並有10秒的前導信號。
{啓始}.{最終}R	將卡帶上之資料讀入{啓始}.{最終}間，若錯誤則顯示"ERR"

執行或表列程式

{位址G}	將控制移轉至以{位址}爲開始的機器語言程式。
{位址L}	將以{位址}爲開始的20個指令反組合並顯示，再按一個L將繼續顯示下20個指令。

迷你組合語言翻譯程式

F666G	進入迷你組合語言翻譯程式 *
${命令}	從迷你組合語言翻譯程式執行監督程式命令。
$FF69G	離開迷你組合語言翻譯程式
{位址}S	反組合，顯示並執行在{位址}之指令，之後顯示6502內部暫存器內含，連續的S命令將連續指令的執行。
{位址}T	一直前進，TRACE命令只有在碰到BRK指令或按 RESET ** 鍵時才停止。

其他命令

I	設定反相顯示模式

* Apple Ⅱ Plus 不適用。

** 自動啓始ROM不適用。

N	設定正常顯示模式
CTRL B	進入目前置於Apple ROM之語言
CTRL C	重新進入目前置於Apple ROM之語言
{數值}+{數值}	將兩數值相加並顯示結果
{數值}-{數值}	將兩數值相減並顯示結果
{漕數} CTRL P	將輸出導至該漕數上的界面板，若{漕數}=0，則被導至Apple顯示幕。
{漕數} CTRL K	從該漕數上的界面板擷取資料，若{漕數}=0，則接受來自Apple鍵盤的資料。
CTRL Y	跳至機器語言副程式$3F8。

一些有用的監督程式副程式

以下列出存於Apple監督程式及自動啓始ROM之有用副程式。在機器語言程式中使用這些程式時，將該副程式所需的適當位址或6502內部暫存器載入適當的值然後執行JSR到副程式之開始位置即可。它將執行該功能並將6502之暫存器依所述設定後回歸。

$FDED　COUT　輸出一個字元

COUT是標準字元輸出副程式，將輸出之字元必置於累加器內，COUT呼叫目前被放於CSW（位置$36及$37）內的字元輸出副程式，一般為COUT1（如下）。

$FDFØ　COUT1　輸出至顯示幕

COUT1將目前存在累加器的內含輸出至Apple顯示幕上目前輸出游標的位置並將該游標前進一位。它依正常/反相位置之設定而輸出字元，它同時處理RETURN 、進線及響鈴字元，回歸時所有暫存器不受影響。

$FE8∅ SETINV 設定反相模式

將COUT1用的視頻輸出設爲反相模式，所有的輸出字元均以白底黑字顯示，Y暫存器被設爲$3F，其他暫存器不變。

$FE84 SETNORM 設定正常模式

將COUT1用的視頻輸出設定爲正常模式，所有輸出字元均以白字黑底顯示，Y暫存器被設爲$FF ，其他不變。

$FD8E CROUT 產生一個RETURN

CROUT送一個RETURN 至目前輸出裝置。

$FD8B CROUT1 RETURN 並清除

CROUT1 將目前游標之後至正文窗之緣的顯示幕清除，然後呼叫CROUT 。

$FDDA PRBYTE 印出十六進制位元組

該副程式將累加器之內含輸出至目前之列表裝置，累加器內含遭破壞。

$FDE3 PRHEX 印出十六進制數位

該副程式將累加器內低尼布之數位以單一十六進制數位方式輸出，累加器內含被破壞。

$F941　　PRNTAX　　　將A與X用十六進制印出

該副程式將A及X之內含以四數位十六進制值印出，A印在前面，X在後面，A之內含一般會被破壞。

$F948　　PRBLNK　　　印出3個空白

將3個空白送出標準輸出裝置，跳回時，A一般為$A0，X為∅。

$F94A　　PRBL2　　　印出很多空白

該副程式將X內之個數的空白送至標準輸出裝置，若X=00則PRBL2將輸出256個空白。

$FF3A　　BELL　　　輸出一"響鈴"字元

該副程式送一個"響鈴"字元至標準輸出裝置，A內含為$87

$FBDD　　BELL1　　　Apple揚聲器"嗶"一聲

該副程式使Apple之揚聲器發出1KHZ之"嗶"聲0.1秒，A及X之內含被破壞。

$FD∅C　　RDKEY　　　得一輸入字元

該副程式為標準字元輸入副程式，在顯示幕輸出游標位置上顯示一閃爍輸入游標，並跳至目前存於KSW（位置$38及$39）之目前輸入副程式，一般為KEYIN（如下）。

$FD35　　RDCHAR　　　得一輸入字元或ESC碼

RDCHAR是從標準輸入獲得字元的另一輸入副程式，但也了解十一個ESCAPE碼（見××頁）。

$FDIB　　KEYIN　　　　讀取 Apple 之鍵盤

鍵盤輸入副程式，讀取 Apple 之鍵盤，等待按鍵並產生亂數種子（見××頁），當鍵被按下時，該副程式移去閃爍游標並將鍵碼回歸至累加器。

$FD6A　　GETLN　　　　得一輸入線及提示符號

GETLN 副程式可收集輸入線（見××頁），程式中可以呼叫該副程式並將適當提示字元放在位置 $33；GETLN 回歸時，輸入線之資料被放於輸入緩衝區（開始位置 $200），而 X 之值爲輸入線上的字元長度。

$FD67　　GETLNZ　　　　得一輸入線

GETLN 是另一個進入 GETLN 的進口點，但在進入 GETLN 之前先送一個 RETURN 至標準輸出。

$FD6F　．GETLN1　　　　得一輸入線組沒有提示符號

GETNN1 是另外一個 GETLN 的進口，但在收集輸入線之前沒有顯示提示符號，但是如果因爲使用太多的後退鍵或 CTRL X 而取消這輸入線的話，GETLN1 將位置 $33 之內含顯示出來並跳至下一線。

$FCA8　　WAIT　　　　遲延

該副程式遲延 A 內數值之時間，並回到呼叫之程式。若 A 之值爲 N，則將遲延 ½（26＋27N＋5N²）μS，回歸時 A 爲 Ø，X 及 Y 不變。

$F864　　SETCOL　　　　設定低解析度繪圖模式

該副程式將低解析度繪圖用的顏色設爲由A送來的顏色，請參閱××頁之低解析度色彩表。

$F85F　XEXTCOL　將色彩加 3

將目前低解析度繪圖用的色彩加 3 。

$F800　PLOT　在低解析度顯示幕上劃一個方塊

該副程式以事先定好的顏色在低解析度顯示幕上劃一個方塊，A 之內含爲方塊之垂直位置，Y 爲水平位址，回歸時 A 被破壞，X 與Y 不受影響。

$F819　HHINE　劃一水平線之方塊

該副程式以事先定好之顏色在低解析度顯示幕上劃一線的方塊，呼叫時 A 放垂直座標而 Y 放水平最左座標，水平線右座標在位置 $2C內，回歸時 A 與 Y 被破壞，X 不變。

$F828　VLINE　劃一垂直線方塊

該副程式以事先定好之顏色在低解析度顯示幕上劃一垂直線方塊，呼叫時水平座標在 Y 內，最頂端之垂直座標在 A 內，而最底垂直座標則在位置 $2D內。A 被破壞。

$F832　CLRSCR　清除整個低解析度顯示幕

該副程式清除整個低解析度顯示幕，如果在正文模式中呼叫 CLRSCR ，則整個顯示幕將出現反相的“ @ ”字元。A與 Y 被破壞。

$F836　CLRTOP　清除低解析度顯示幕之頂端

CLRTOP 之動作與 CLRSCR 相同，只是它僅清除顯示幕之

頂端40橫列。

$F871　SCRN　讀取低解析度顯示幕

該副程式回歸時帶回低解析度顯示幕某方塊之顏色碼，呼叫法與PLOT一樣（見上所述），顏色碼將置於A內，其它暫存器不變。

$FB1E　PREAD　讀取遊戲控制器

PREAD將帶回代表遊戲控制器位置之值，呼叫時X放控制器代號（0至3），如果代號不合，將有奇怪現象發生，回歸時Y之內含爲$00至$FF，A則被破壞。

$FF2D　PRERR　印出"ERR"

將"ERR"及響鈴送至標準輸出裝置，A被破壞。

$FF4A　IOSAVE　儲存所有暫存器

將所有6502內部暫存器存於$45至$49之位置內，順序爲A－X－Y－P－S，A及X被改變，十進模式被清除。

$FF3F　IOREST　取回所有暫存器

將位置$45至$49之內含載入6502內部暫存器內。

監督程式的特殊位置

表 14 ：第三頁監督程式位置

位址		使用	
十進制	十六進制	監督程式 ROM	自動啓始 ROM
1008 1009	$3F0 $3F1	無	存放處理機器語言" BRK "要求副程式的位址，一般爲$FA59
1010 1011	$3F2 $3F3	無	軟性進口向量
1012	$3F4	無	開機位元組
1013 1014 1015	$3F5 $3F6 $3F7	存放處理Applesoft Ⅱ " & "命令之副程式的跳越指令，一般爲$4C $58 $FF	
1016 1017 1018	$3F8 $3F9 $3FA	存放處理使用者(CTRL Y)命令副程式的跳越指令	
1019 1020 1021	$3FB $3FC $3FD	存放處理不可遮擋中斷副程式的跳越指令	
1022 1023	$3FE $3FF	存放處理中斷要求副程式的位址	

迷你組合語言翻譯程式之指令格式

Apple迷你組合語言翻譯程式可認識6502組合語言程式的56個縮碼及13種定址型式。縮碼是標準型式，如MOS Technology / Synertek 6500 Programming Manual（Apple編號A2L0003）之方式一致，但定址型式略有不同，以下是6502組合語言Apple標準定址形式之格式：

表15　迷你組合語言翻譯程式定址格式

型式	格式
累加器	無
立即	＃$｛數值｝
絕對	$｛位址｝
零頁	$｛位址｝
指標零頁	$｛位址｝，X $｛位址｝，Y
指標絕對	$｛位址｝，X $｛位址｝，Y
隱含	無
相對	$｛位址｝
指標間接	（$｛位址｝，X）
間接指標	（$｛位址｝），Y
絕對間接	（$｛位址｝）

｛位址｝由一個或更多十六進制數位組成，迷你組合語言翻譯程式對位址之解釋法與監督程式同：即如果位址少於4個，前頭補零；若多於四個，僅用最後四個。"$"符號表示十六進制，迷你

組合語言翻譯程式忽略它，因此可以不用。

在絕對與零頁定址型式中，沒有語法上之不同，在可以同時使用絕對與零頁地址型式的指令中，若給予的位址大於 $FF則迷你組合語言翻譯程式自動選用絕對定址形式，若位址小於$100，則採零頁定址模式。

累加器及隱含式定址型式不需運算子。

分支指令，使用相對定址模式，需要分支的目標位址。迷你組合語言翻譯程式可以自動地算出分支的相對距離，如果至該分支目標位址之距離大於 127，迷你組合語言翻譯程式將發出"嗶"並在目標位址下顯示抑揚符號（ ^ ）並丟棄該線。

如果給予迷你組合語言翻譯程式指令縮碼、運算子，但運算子之定址型式並不適於該指令，則迷你組合語言翻譯程式將不接受該輸入線。

<table>
<tr><th colspan="2">系統記憶器分配圖</th></tr>
<tr><td>頁　　數：
十進制　十六進制</td><td></td></tr>
<tr><td>0　$00
1　$01
2　$02
⋮　⋮
190　$BE
191　$BF</td><td>RAM（48K）</td></tr>
<tr><td rowspan="2">192　$C0
193　$C1
⋮　⋮
198　$C6
199　$C7
200　$C8
201　$C9
⋮　⋮
206　$CE
207　$CF</td><td>I/O（2K）</td></tr>
<tr><td>I/O ROM
（2K）</td></tr>
<tr><td>208　$D0
209　$D1
⋮　⋮
254　$FE
255　$FF</td><td>ROM（12K）</td></tr>
</table>

圖 5　系統記憶器分配圖

表 16　RAM之結構及使用

<table>
<tr><th colspan="2">頁　數</th><th colspan="2" rowspan="2">用　　　於</th></tr>
<tr><th>十進制</th><th>十六進制</th></tr>
<tr><td>0</td><td>$ 00</td><td colspan="2">系統程式</td></tr>
<tr><td>1</td><td>$ 01</td><td colspan="2">系統堆疊</td></tr>
<tr><td>2</td><td>$ 02</td><td colspan="2">GETLN輸入緩衝區</td></tr>
<tr><td>3</td><td>$ 03</td><td colspan="2">監督程式向量位置</td></tr>
<tr><td>4
5
6
7</td><td>$ 04
$ 05
$ 06
$ 07</td><td colspan="2">正文及低解析度繪圖第一頁儲存區</td></tr>
<tr><td>8
9
10
11</td><td>$ 08
$ 09
$ 0A
$ 0B</td><td>正文及低解析度繪圖第二頁儲存區</td><td rowspan="5">自　由

RAM</td></tr>
<tr><td>12 至 31</td><td>$ 0C
$ 1F</td><td></td></tr>
<tr><td>32 至 63</td><td>$ 20
$ 3F</td><td>高解析度繪圖第一頁儲存區</td></tr>
<tr><td>64 至 95</td><td>$ 40
$ 5F</td><td>高解析度繪圖第二頁儲存區</td></tr>
<tr><td>96 至 191</td><td>$ 60
$ BF</td><td></td></tr>
</table>

<table>
<tr><th colspan="4">表17　ROM之結構及使用</th></tr>
<tr><th colspan="2">頁　數</th><th rowspan="2" colspan="2">用　於</th></tr>
<tr><th>十進制</th><th>十六進制</th></tr>
<tr><td>208</td><td>$D0</td><td rowspan="2">Programmer's Aid #1</td><td rowspan="10">Applesoft II BASIC</td></tr>
<tr><td>212</td><td>$D4</td></tr>
<tr><td>216</td><td>$D8</td><td rowspan="2"></td></tr>
<tr><td>220</td><td>$DC</td></tr>
<tr><td>224</td><td>$E0</td><td rowspan="5">Integer BASIC</td></tr>
<tr><td>228</td><td>$E4</td></tr>
<tr><td>232</td><td>$E8</td></tr>
<tr><td>236</td><td>$EC</td></tr>
<tr><td>240</td><td>$F0</td></tr>
<tr><td>244</td><td>$F4</td><td>實用副程式</td></tr>
<tr><td>248</td><td>$F8</td><td rowspan="2">監督程式</td><td rowspan="2">自動啓始程式</td></tr>
<tr><td>252</td><td>$FC</td></tr>
</table>

零頁記憶器分配圖

表 18　監督程式零頁使用情形

十進制	十六進制	0 $0	1 $1	2 $2	3 $3	4 $4	5 $5	6 $6	7 $7	8 $8	9 $9	10 $A	11 $B	12 $C	13 $D	14 $E	15 $F
0	$00																
16	$10																
32	$20	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
48	$30	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
64	$40	•	•	•	•	•	•	•	•	•	•					•	•
80	$50	•	•	•	•	•	•										
96	$60																
112	$70																
128	$80																
144	$90																
160	$A0																
176	$B0																
192	$C0																
208	$D0																
224	$E0																
240	$F0																

表 19　Applesoft II BASIC零頁使用情形

十進制	十六進制	0 $0	1 $1	2 $2	3 $3	4 $4	5 $5	6 $6	7 $7	8 $8	9 $9	10 $A	11 $B	12 $C	13 $D	14 $E	15 $F
0	$00	•	•	•	•	•	•					•	•	•	•	•	•
16	$10	•	•	•	•	•	•	•	•	•							
32	$20																
48	$30																
64	$40																
80	$50	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
96	$60	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
112	$70	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
128	$80	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
144	$90	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
160	$A0	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
176	$B0	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
192	$C0	•	•	•	•	•	•	•	•	•	•	•	•	•	•		
208	$D0	•	•	•	•	•	•			•	•	•	•	•	•	•	•
224	$E0	•	•	•		•	•	•	•	•	•	•					
240	$F0	•	•	•	•	•	•	•	•	•							

表 20 Apple DOS 3.2 零頁使用情形																	
十進制	十六進制	0 $0	1 $1	2 $2	3 $3	4 $4	5 $5	6 $6	7 $7	8 $8	9 $9	10 $A	11 $B	12 $C	13 $D	14 $E	15 $F
0	$ 00																
16	$ 10																
32	$ 20							•	•			•	•	•	•	•	•
48	$ 30						•	•	•	•	•					•	•
64	$ 40	•	•	•	•	•	•	•	•	•		•	•	•	•		
80	$ 50																
96	$ 60								•	•	•	•					•
112	$ 70	•															
128	$ 80																
144	$ 90																
160	$A0																•
176	$B0	•															
192	$C0											•	•	•	•		
208	$D0									•							
224	$E0																
240	$F0																

表 21 Integer BASIC 零頁使用情形																	
十進制	十六進制	0 $0	1 $1	2 $2	3 $3	4 $4	5 $5	6 $6	7 $7	8 $8	9 $9	10 $A	11 $B	12 $C	13 $D	14 $E	15 $F
0	$ 00																
16	$ 10																
32	$ 20																
48	$ 30																
64	$ 40											•	•	•	•		
80	$ 50						•	•	•	•	•	•	•	•	•	•	•
96	$ 60	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
112	$ 70	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
128	$ 80	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
144	$ 90	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
160	$A0	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
176	$B0	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
192	$C0	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
208	$D0	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
224	$E0																
240	$F0																

下列之符號將應用於本總結中：

A　　：累加器
X，Y：指標暫存器
M　　：記憶器
C　　：借位
P　　：處理機狀態暫存器
S　　：堆疊指示器
√　　：改變
—　　：未改變
+　　：加
∧　　：邏輯 AND
−　　：減
∀　　：邏輯互斥 OR
↑　　：從堆疊中取出
↓　　：堆至堆疊上
→　　：傳至
←　　：傳至
V　　：邏輯 OR
PC　：程式計數器
PCH：程式計數器之高位元組
PCL：程式計數器之低位元組
OPER：運算子
#　　：立即定址型式

圖 1　ASL：向左移一位元之動作

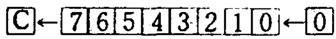


圖 2　ROL：向左旋轉一位元（記憶器或累加器）

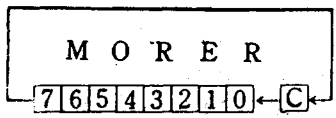
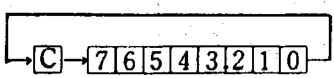


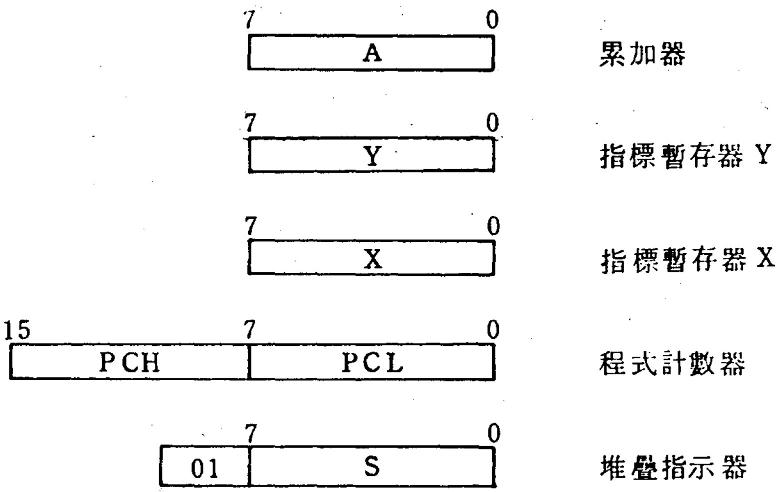
圖 3　ROR：向右旋轉一位元



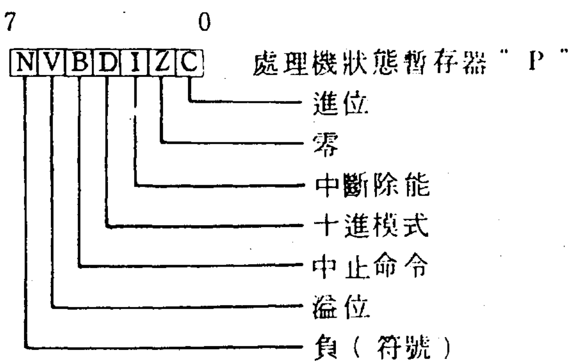
注

注意 1：BIT：測試各位元，位元 6 及 7 被傳送至狀態暫存器，若 A ∧ M 之結果爲 0，則 Z = 1，否則 Z = 0。

規劃模式



7
0
A
累加器
7
0
Y
指標暫存器 Y
7
0
X
指標暫存器 X
15
7
0
PCH
PCL
程式計數器
7
0
01
S
堆疊指示器



7
0
N V B D I Z C
處理機狀態暫存器 " P "
進位
零
中斷除能
十進模式
中止命令
溢位
負（符號）

ADC 與進位值一起加至累加器

NZCIDV
✓✓✓--✓

運算：A＋M＋C→A，C

定址法	組合語言形式	運算碼	位元組數	週期數
立即	ADC #dd	69	2	2
零頁	ADC aa	65	2	3
零頁，X	ADC aa，X	75	2	4
絕對	ADC aaaa	6D	3	4
絕對，X	ADC aaaa，X	7D	3	4*
絕對，Y	ADC aaaa，Y	79	3	4*
（間接，X）	ADC （aa，X）	61	2	6
（間接），Y	ADC （aa），Y	71	2	5*

* 若超出頁界限，則加1。

AND 記憶位置內含與累加器內含AND

NZCIDV
✓✓----

運算：A∧M→A

定址法	組合語言形式	運算碼	位元組數	週期數
立即	AND #dd	29	2	2
零頁	AND aa	25	2	3
零頁，X	AND aa，X	35	2	4
絕對	AND aaaa	2D	3	4
絕對，X	AND aaaa，X	3D	3	4*
絕對，Y	AND aaaa，Y	39	3	4*
（間接，X）	AND （aa，X）	21	2	6
（間接），Y	AND （aa），Y	31	2	5*

* 超越頁界限時加1。

ASL　　　　　　　　一累加器內含左移

運算：C ← [7|6|5|4|3|2|1|0] ← 0

NZCIDV
√√√−−−

定址法	組合語言形式		運算碼	位元組數	週期數
累加器	ASL	A	0A	1	2
零頁	ASL	aa	06	2	5
零頁，X	ASL	aa，X	16	2	6
絕对	ASL	aaaa	0E	3	6
絕對，X	ASL	aaaa，X	1E	3	7

BCC　　　　　　　　一進位清除時跳越

AZCIDV
−−−−−−

運算：c = 0 時跳越

定址法	組合語言形式		運算碼	位元組數	週期數
相對	BCC	aa	90	2	2*

* 跳越至同頁時加 1 ，跳至不同頁時加 2 。

附註：AIM65 能接受BCC　aaaa　之絕對定址形式，並將之轉換成相對位址。

BCS　　　　　　　　一進位置定時跳越

NZCIDV
−−−−−−

運算：c = 1 時跳越.。

定址法	組合語言形式		運算碼	位元組數	週期數
相對	BCS	aa	BO	2	2*

* 跳越至同頁時加 1 ，跳至次頁時加 2 。

註：AIM65 可接受絕對定址之運算元，並將之換成相對位址。

BEQ —結果等於零時跳越

NZCIDV

運算：Z = 1 時跳越發生

定址法	組合語言形式	運算碼	位元組數	週期數
相對	BEQ　　aa	F0	2	2*

* 跳至同頁時加 1，跳至次一頁時加 2。

註：AIM65 可接受絕對位址之運算元（形如 BEQ　aaaa），並將之轉換成相對位址。

BIT —測試記憶位元值

運算：$A \wedge M$，$M_7 \rightarrow N$，$M_6 \rightarrow V$

第 6 與 7 位元抄至狀態暫存器。　　NZCIDV

若 $A \wedge M$ 結果為零，則 Z = 1；否則 Z = 0。　　M_7 ✓ - - - M_6

定址法	組合語言形式	運算碼	位元組數	週期數
零頁	BIT　　aa	24	2	3
絕對	BIT　　aaaa	2C	3	4

BMI —運算結果負時跳越

NZCIDV

運算：N = 1 時，跳越發生。

定址法	組合語言形式	運算碼	位元組數	週期數
相對	BMI　　aa	30	2	2*

* 跳至同頁時加 1，跳至不同頁時加 2。

註：AIM 65 能接受絕對位址，並將之轉換成一相對位址。

BNE

—結果不等於零時跳越

NZCIDV

運算：Z = 0 時，跳越發生。

定址法	組合語言形式	運算碼	位元組數	週期數
相對	BNE aa	D0	2	2 *

* 跳越至同頁時加 1 ，跳越至不同頁時加 2 。

註：AIM 65 能接受一絕對位址，並將之轉換成相對位址。

BPL

—結果為正時跳越

NZCIDV

運算：N = 0 時，跳越發生。

定址法	組合語言形式	運算碼	位元組數	週期數
相對	BPL aa	10	2	2 *

* 跳越至同頁時加 1 ，跳越至不同頁時加 2 。

註：AIM 65 能接受一絕對位址之運算元（指令格式BPL aaaa），並將之轉換成相對位址。

BRK

—迫使中斷

BNZCIDV
1---1--

運算：被迫插斷PC+2↓P↓

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	BRK	00	1	7

BVC　　一溢位清除時跳越

NZCIDV

運算：V = 0 時，跳越發生。

定址法	組合語言形式	運算碼	位元組數	週期數
相對	BVC　aa	50	2	2*

* 跳越至同頁時加 1，跳越至不同頁時加 2。

註：AIM 65 能接受一絕對位址，並將之轉換成相對位址。

BVS　　一溢位旗號置定時跳越

NZCIDV

運算：V = 1 時，跳越發生。

定址法	組合語言形式	運算碼	位元組數	週期數
相對	BVS　aa	70	2	2*

* 跳越至同頁時加 1，跳越至不同頁時加 2。

註：AIM 65 能接受一絕對位址，並將之轉換成相對位址。

CLC　　一清除進位旗號

NZCIDV
--0---

運算：0 → C

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	CLC	18	1	2

CLD

—清除十進制位元

NZCIDV
----0-

運算：0 → D

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	CLD	D8	1	2

CLI

—清除插斷禁能位元

NZCIDV
---0--

運算：0 → I

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	CLI	58	1	2

CLV

—清除溢位旗號

NZCIDV
-----0

運算：0 → V

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	CLV	B8	1	2

CMP — 記憶位置內含與累加器內含相比

NZCIDV
✓✓✓---

運算：A－M

定址法	組合語言形式	運算碼	位元組數	週期數
立即	CMP #dd	C9	2	2
零頁	CMP aa	C5	2	3
零頁，X	CMP aa，X	D5	2	4
絕對	CMP aaaa	CD	3	4
絕對，X	CMP aaaa，X	DD	3	4*
絕對，Y	CMP aaaa，Y	D9	3	4*
(間接，X)	CMP (aa，X)	C1	2	6
(間接)，Y	CMP (aa)，Y	D1	2	5*

* 若越過頁邊界，則加1。

CPX —記憶位置內含與X暫存器內含相比

NZCIDV
✓✓✓---

運算：X－M

定址法	組合語言形式	運算碼	位元組數	週期數
立即	CPX #dd	E0	2	2
零頁	CPX aa	E4	2	3
絕對	CPX aaaa	EC	3	4

CPY —記憶位置內含與Y暫存器內含相比

NZCIDV
✓✓✓---

運算：Y－M

定址法	組合語言形式	運算碼	位元組數	週期數
立即	CPY #dd	C0	2	2
零頁	CPY aa	C4	2	3
絕對	CPY aaaa	CC	3	4

DEC
－記憶位置內含值減－

NZCIDV
√√－－－－

運算：M－1→M

定址法	組合語言形式	運算碼	位元組數	週期數
零頁	DEC aa	C6	2	5
零頁，X	DEC aa,X	D6	2	6
絕對	DEC aaaa	CE	3	6
絕對，X	DEC aaaa,X	DE	3	7

DEX
－X暫存器內含值減－

NZCIDV
√√－－－－

運算：X－1→X

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	DEX	CA	1	2

DEY
－Y暫存器內含值減－

NZCIDV
√√－－－－

運算：Y－1→Y

定址法	語言組合形式	運算碼	位元組數	週期數
隱含	DEY	88	1	2

EOR

—記憶位置值與累加器值EOR

NZCIDV
√√----

運算：AVM→A

定址法	組合語言形式	運算碼	位元組數	週期數
立即	EOR #dd	49	2	2
零頁	EOR aa	45	2	3
零頁，X	EOR aa,x	55	2	4
絕對	EOR aaaa	4D	3	4
絕對，X	EOR aaaa,X	5D	3	4*
絕對，Y	EOR aaaa,Y	59	3	4*
(間接，X)	EOR (aa,X)	41	2	6
(間接)，Y	EOR (aa),Y	51	2	5*

INC

—記憶位置內含值加—

NZCIDV
√√----

運算：M+1→M

定址法	組合語言形式	運算碼	位元組數	週期數
零頁	INC aa	E6	2	5
零頁，X	INC aa,X	F6	2	6
絕對	INC aaaa	EE	3	6
絕對，X	INC aaaa,X	FE	3	7

INX

—X暫存器內含值加—

NZCIDV
√√----

運算：X+1→X

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	INX	E8	1	2

INY　　—Y暫存器內含值加一

NZCIDV
√√----

運算：Y＋1→Y

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	INY	C8	1	2

JMP　　—跳越

運算：(PC+1)→PCL　　NZCIDV
(PC+2)→PCH　　------

定址法	組合語言形式	運算碼	位元組數	週期數
絕對	JMP aaaa	4C	3	3
間接	JMP (aaaa)	6C	3	5

JSR　　—跳越至副程式

運算：PC+2↓，(PC+1)→PCL
(PC+2)→PCH

NZCIDV

定址法	組合語言形式	運算碼	位元組數	週期數
絕對	JSR aaaa	20	3	6

LDA　　一記憶位置內含取入累加器

NZCIDV

運算：M→A　　√√----

定址法	組合語言形式	運算碼	位元組數	週期數
立即	LDA #dd	A9	2	2
零頁	LDA aa	A5	2	3
零頁，X	LDA aa,X	B5	2	4
絕對	LDA aaaa	AD	3	4
絕對，X	LDA aaaa,X	BD	3	4*
絕對，Y	LDA aaaa,Y	B9	3	4*
(間接，X)	LDA (aa,X)	A1	2	6
(間接),Y	LDA (aa),Y	B1	2	5*

* 若超越頁邊界，則加1。

LDX　　一記憶位置內含取入X暫存器

NZCIDV

運算：M→X　　√√----

定址法	組合語言形式	運算碼	位元組數	週期數
立即	LDX #dd	A2	2	2
零頁	LDX aa	A6	2	3
零頁，Y	LDX aa,Y	B6	2	4
絕對	LDX aaaa	AE	3	4
絕對，Y	LDX aaaa,Y	BE	3	4*

* 若越過頁邊界，則加1。

LDY —記憶位置內含值取入Y暫存器

NZCIDV
√√----

運算：M→Y

定址法	組合語言形式	運算碼	位元組數	週期數
立即	LDY #dd	A0	2	2
零頁	LDY aa	A4	2	3
零頁，X	LDY aa,X	B4	2	4
絕對	LDY aaaa	AC	3	4
絕對，X	LDY aaaa,X	BC	3	4*

*越過頁界限時加1。

LSR —邏輯右移

NZCIDV
0√√---

運算：0→[7|6|5|4|3|2|1|0]→C

定址法	組合語言形式	運算碼	位元組數	週期數
累加器	LSR A	4A	1	2
零頁	LSR aa	46	2	5
零頁，X	LSR aa,X	56	2	6
絕對	LSR aaaa	4E	3	6
絕對，X	LSR aaaa,X	5E	3	7

NOP —無運算

運算：無運算

NZCIDV

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	NOP	EA	1	2

ORA

一記憶位置內含與累加器內含OR

NZCIDV
√√----

運算：A V M→A

定址法	組合語言形式	運算碼	位元組數	週期數
立即	ORA #dd	09	2	2
零頁	ORA aa	05	2	3
零頁，X	ORA aa,X	15	2	4
絕對	ORA aaaa	0D	3	4
絕對，X	ORA aaaa,X	1D	3	4*
絕對，Y	ORA aaaa,Y	19	3	4*
(間接，X)	ORA (aa,X)	01	2	6
(間接)，Y	ORA (aa),Y	11	2	5*

* 超越頁界時加1。

PHA

一累加器內含推入堆疊器

NZCIDV

運算：A ↓

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	PHA	48	1	3

PHP

一狀態暫存器內含推入堆疊器

NZCIDV

運算：P ↓

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	PHP	08	1	3

PLA

一自堆疊器拉取至累加器

NZCIDV

運算：A ↑　　　　✓✓----

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	PLA	68	1	4

PLP

一狀態暫存器內含自堆疊器拉回

NZCIDV

運算：P ↑　　　　來自堆疊內含

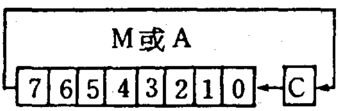
定址法	組合語言形式	運算碼	位元組數	週期數
隱含	PLP	28	1	4

ROL

一左旋轉

NZCIDV

運算：



✓✓✓---

定址法	組合語言形式	運算碼	位元組數	週期數
累加器	ROL　A	2A	1	2
零頁	ROL　aa	26	2	5
零頁，X	ROL　aa,X	36	2	6
絕對	ROL　aaaa	2E	3	6
絕對，X	ROL　aaaa,X	3E	3	7

ROR —右旋轉

NZCIDV
✓✓✓---

運算：

定址法	組合語言形式	運算碼	位元組數	週期數
累加器	ROR A	6A	1	2
零頁	ROR aa	66	2	5
零頁，X	ROR aa,X	76	2	6
絕對	ROR aaaa	6E	3	6
絕對，X	ROR aaaa,X	7E	3	7

RTI —插斷回返

NZCIDV

運算：P↑PC↑ 來自堆疊內含

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	RTI	40	1	6

RTS —副程式回返

NZCIDV

運算：PC↑，PC+1→PC

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	RTS	60	1	6

SBC —累加器值減去記憶位置值與進位

運算：$A - M - \bar{C} \rightarrow A$ NZCIDV

附註：$\bar{C}$ = 借位 ✓✓✓−− ✓

定址法	組合語言形式	運算碼	位元組數	週期數
立即	SBC #dd	E9	2	2
零頁	SBC aa	E5	2	3
零頁，X	SBC aa,X	F5	2	4
絕對	SBC aaaa	ED	3	4
絕對，X	SBC aaaa,X	FD	3	4*
絕對，Y	SBC aaaa'Y	F9	3	4*
（間接，X）	SBC (aa,X)	E1	2	6
（間接），Y	SBC (aa),Y	F1	2	5*

* 若超越頁界限，則加 1 。

SEC —進位旗號置定爲1

NZCIDV
--1---

運算：$1 \rightarrow C$

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	SEC	38	1	2

SED —置定爲十進制

NZCIDV
----1-

運算：$1 \rightarrow D$

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	SED	F8	1	2

SEI　　　　　—置定插斷禁能狀態

NZCIDV

運算：1→I　　　　　　---1--

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	S E I	78	1	2

STA　　　　　—累加器內含存出至記憶器

NZCIDV

運算：A→M　　　　　　------

定址法	組合語言形式	運算碼	位元組數	週期數
零頁	STA aa	85	2	3
零頁，X	STA aa,X	95	2	4
絕對	STA aaaa	8D	3	4
絕對，X	STA aaaa,X	9D	3	5
絕對，Y	STA aaaa,Y	99	3	5
（間接，X）	STA (aa,X)	81	2	6
（間接），Y	STA (aa),Y	91	2	6

STX　　　　　—X暫存器內含存出記憶器

NZCIDV

運算：X→M　　　　　　------

定址法	組合語言形式	運算碼	位元組數	週期數
零頁	STX aa	86	2	3
零頁，Y	STX aa,Y	96	2	4
絕對	STX aaaa	8E	3	4

STY —Y 暫存器內含存出記憶器

NZCIDV

運算：Y→M

定址法	組合語言形式	運算碼	位元組數	週期數
零頁	STY aa	84	2	3
零頁，X	STY aa，X	94	2	4
絕對	STY aaaa	8C	3	4

TAX —累加器內含抄至X暫存器

NZCIDV
√√----

運算：A→X

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	TAX	AA	1	2

TAY —累加器內含抄至Y暫存器

NZCIDV
√√----

運算：A→Y

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	TAY	A8	1	2

TSX —堆疊指示器內含抄至X暫存器

NZCIDV

運算：S→X　　✓✓----

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	TSX	BA	1	2

TXA —X暫存器內含抄至累加器

NZCIDV

運算：X→A　　✓✓----

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	TXA	8A	1	2

TXS —X暫存器內含抄至堆疊指示器

NZCIDV

運算：X→S　　------

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	TXS	9A	1	2

TYA —Y暫存器內含抄至累加器

NZCIDV

運算：Y→A　　✓✓----

定址法	組合語言形式	運算碼	位元組數	週期數
隱含	TYA	98	1	2

表 1 ：鍵盤特殊位置			
位置			描述
十六進制	十進制		
$CØØØ	49152	-16384	鍵盤資料
$CØ1Ø	49168	-16368	清除鍵盤激勵

表 4 ：視頻顯示器記憶器範圍					
顯示幕	頁	開始		結束	
		十六進制	十進制	十六進制	十進制
正文／低解析度	第一頁	$4ØØ	1Ø24	$7FF	2Ø47
	第二頁	$8ØØ	2Ø48	$BFF	3Ø71
高解析度	第一頁	$2ØØØ	8192	$3FFF	16383
	第二頁	$4ØØØ	16384	$5FFF	24575

表 5 ：顯示幕軟性開關			
位置			描述
十六進制	十進制		
$CØ5Ø	49232	-163Ø4	顯示一種繪圖模式
$CØ51	49233	-163Ø3	顯示正文模式
$CØ52	49234	-163Ø2	顯示所有正文或繪圖模式
$CØ53	49235	-163Ø1	正文與繪圖模式混合
$CØ54	49236	-163ØØ	顯示第一頁
$CØ55	49237	-16299	顯示第二頁
$CØ56	49238	-16298	顯示低解析度繪圖模式
$CØ57	49239	-16297	顯示高解析度繪圖模式

表 9 ：告示特殊位置				
告示	狀態	位置		
		十進制		十六進制
Ø	"Ø"	4924Ø	-16296	$CØ58
	"1"	49241	-16295	$CØ59
1	"Ø"	49242	-16294	$CØ5A
	"1"	49243	-16293	$CØ5B
2	"Ø"	49244	-16292	$CØ5C
	"1"	49245	-16291	$CØ5D
3	"Ø"	49246	-1629Ø	$CØ5E
	"1"	49247	-16289	$CØ5F

表 10　輸入／出特殊位置

功能	位置			讀／寫
	十進制		十六進制	
揚聲器	492ØØ	-16336	$CØ3Ø	讀
卡帶輸出	49184	-16352	$CØ2Ø	讀
卡帶輸入	49256	-16288	$CØ6Ø	讀
告示輸出	4924Ø 至 49247	-16296 至 -16289	$CØ58 至 $CØ5F	讀／寫
旗標輸入	49249	-16287	$CØ61	讀
	4925Ø	-16286	$CØ62	讀
	49251	-16285	$CØ63	讀
類比輸入	49252	-16284	$CØ64	讀
	49253	-16283	$CØ65	
	49254	-16282	$CØ66	
	49255	-16281	$CØ67	
類比清除	49264	-16272	$CØ7Ø	讀／寫
實用激勵	49216	-1632Ø	$CØ4Ø	讀

表 11 ：正文窗特殊位置

功能	位置		最小／正常／最大值	
	十進制	十六進制	十進制	十六進制
左緣	32	$2Ø	Ø/Ø/39	$Ø/$Ø/$17
寬	33	$21	Ø/4Ø/4Ø	$Ø/$28/$28
上緣	34	$22	Ø/Ø/24	$Ø/$Ø/$18
下緣	35	$23	Ø/24/24	$Ø/$18/$18

表12：正常/反相控制值

值		效　　果
十進制	十六進制	
255	$FF	COUT 將依正常模式顯示字元
63	$3F	COUT 將依反相模式顯示字元
127	$7F	COUT 將依閃爍模式顯示字母而依反相模式顯示其他字元

表13：自動啓始ROM特殊位置

位　　置		內　　含
十　進　制	十六進制	
1∅1∅ 1∅11	$3F2 $3F3	軟性進口向量，該兩位置包含任何語言的進口點位址，一般爲$E003
1∅12	$3F4	開機位元組，一般爲$45
64367 (-1169)	$FB6F	該處爲機器語言設定開機位元組的位置

表 14：第三頁監督程式位置			
位址		使用	
十進制	十六進制	監督程式 ROM	自動啓始 ROM
1øø8 1ø ø9	$3Fø $3F1	無	存放處理機器語言"BRK"要求副程式的位址，一般爲$FA59
1ø1ø 1ø11	$3F2 $3F3	無	軟性進口向量
1ø12	$3F4	無	開機位元組
1ø13 1ø14 1ø15	$3F5 $3F6 $3F7	存放處理Applesoft Ⅱ "&" 命令之副程式的跳越指令，一般爲$4C $58 $FF	
1ø16 1ø17 1ø18	$3F8 $3F9 $3FA	存放處理使用者(CTRL Y)命令副程式的跳越指令	
1ø19 1ø2ø 1ø21	$3FB $3FC $3FD	存放處理不可遮擋中斷副程式的跳越指令	
1ø22 1ø23	$3FE $3FF	存放處理中斷要求副程式的位址	

表 22：內造式 I／O 位置

	$Ø	$1	$2	$3	$4	$5	$6	$7	$8	$9	$A	$B	$C	$D	$E	$F
$CØØØ	鍵盤資料輸入															
$CØ1Ø	清除鍵盤激勵															
$CØ2Ø	卡帶輸出連續反相（軟性開關）															
$CØ3Ø	揚聲器連續反相（軟性開關）															
$CØ4Ø	實用激勵															
$CØ5Ø	gr	tx	nomix	mix	pri	sec	lores	hires	anØ		an1		an2		an3	
$CØ6Ø	cin	pb1	pb2	pb3	gcØ	gc1	gc2	gc3	重覆$C060～$C067							
$CØ7Ø	遊戲控制器激勵															

縮寫之意義：

gr　　：設定繪圖模式
tx　　：設定正文模式
nomix：設定全部正文或繪圖
mix　：正文及繪圖混合
pri　：顯示第一頁
sec　：顯示第二頁
lores：顯示低解析度繪圖
hires：顯示高解析度繪圖
an　：告示輸出
pb　：按扭輸入
gc　：遊戲控制器輸入
cin　：卡帶輸入

表 23：周邊擴充板 I／O位置

	$Ø	$1	$2	$3	$4	$5	$6	$7	$8	$9	$A	$B	$C	$D	$E	$F
$CØ8Ø																
$CØ9Ø																
$CØAØ																
$CØBØ																
$CØCØ																
$CØDØ																
$CØEØ																
$CØFØ																

槽數 {0, 1, 2, 3, 4, 5, 6, 7} 之輸入／輸出

表 24：周邊擴充板PROM位置

	$ØØ	$1Ø	$2Ø	$3Ø	$4Ø	$5Ø	$6Ø	$7Ø	$8Ø	$9Ø	$AØ	$BØ	$CØ	$DØ	$EØ	$FØ
$C1ØØ																
$C2ØØ																
$C3ØØ																
$C4ØØ																
$C5ØØ																
$C6ØØ																
$C7ØØ																

槽數 {1, 2, 3, 4, 5, 6, 7} 之PROM預留位址

表25：I／O位置基礎位置

基礎位置	濽數 Ø	1	2	3	4	5	6	7
$CØ8Ø	$CØ8Ø	$CØ9Ø	$CØAØ	$CØBØ	$CØCØ	$CØDØ	$CØEØ	$CØFØ
$CØ81	$CØ81	$CØ91	$CØA1	$CØB1	$CØC1	$CØD1	$CØE1	$CØF1
$CØ82	$CØ82	$CØ92	$CØA2	$CØB2	$CØC2	$CØD2	$CØE2	$CØF2
$CØ83	$CØ83	$CØ93	$CØA3	$CØB3	$CØC3	$CØC3	$CØE3	$CØF3
$CØ84	$CØ84	$CØ94	$CØA4	$CØB4	$CØC4	$CØD4	$CØE4	$CØF4
$CØ85	$CØ85	$CØ95	$CØA5	$CØB5	$CØC5	$CØD5	$CØE5	$CØF5
$CØ86	$CØ86	$CØ96	$CØA6	$CØB6	$CØC5	$CØD6	$CØE6	$CØF6
$CØ87	$CØ87	$CØ97	$CØA7m	$CØB7	$CØC7	$CØD7	$CØE7	$CØF7
$CØ88	$CØ88	$CØ98	$CØA8	$CØB8	$CØC8	$CØD8	$CØE8	$CØF8
$CØ98	$CØ98	$CØ99	$CØ9A	$CØB9	$CØC9	$CØD9	$CØE9	$CØF9
$CØ8A	$CØ8A	$CØ9A	$CØAA	$CØBA	$CØCA	$CØDA	$CØEA	$CØFA
$CØ8B	$CØ8B	$CØ9B	$CØAB	$ CBB	$CØCB	$CØDB	$CØEB	$CØFB
$CØ8C	$CØ8C	$CØ9C	$CØAC	$CØBC	$CØCC	$CØDC	$CØEC	$CØFC
$CØ8D	$CØ8D	$CØ9D	$CØAD	$CØBD	$CØCD	$CØDD	$CØED	$CØFD
$CØ8E	$CØ8E	$CØ9E	$CØAE	$CØBE	$CØCE	$CØDE	$CØEE	$CØFE
$CØ8F	$CØ8F	$CØ9F	$CØAF	$CØBF	$CØCF	$CØDF	$CØEF	$CØFF
				I／O位置				

表26：I／O隨意書寫RAM位置

基礎位址	濽數 1	2	3	4	5	6	7
$Ø478	$Ø479	$Ø47A	$Ø47B	$Ø47C	$Ø47D	$Ø47E	$Ø47F
$Ø4F8	$Ø4F9	$Ø4FA	$Ø4FB	$Ø4FC	$Ø4FD	$Ø4FE	4Ø4FF
$Ø578	$Ø579	$Ø57A	$Ø57B	$Ø57C	$Ø58D	$Ø57E	$Ø57F
$Ø5F8	$Ø5F9	$Ø5FA	$Ø5FB	$Ø5FC	$Ø5FD	$Ø5FE	$Ø5FF
$Ø678	$Ø679	$Ø67A	$Ø67B	$Ø67C	$Ø67D	$Ø67E	$Ø67F
$Ø6F8	$Ø6F9	$Ø6FA	$Ø6FB	$Ø6FC	$Ø6FD	$Ø6FE	$Ø6FF
$Ø778	$Ø779	$Ø77A	$Ø77B	$Ø77C	$Ø77D	$Ø77E	$Ø77F
$Ø7F8	$Ø7F9	$Ø7FA	$Ø7FB	$Ø7FC	$Ø8FD	$Ø7FE	$Ø7FF

十六進位數操作碼

00 — BRK
01 — ORA — (Indirect, X)
02 — NOP
03 — NOP
04 — NOP
05 — ORA — Zero Page
06 — ASL — Zero Page
07 — NOP
08 — PHP
09 — ORA — Immediate
0A — ASL — Accumulator
0B — NOP
0C — NOP
0D — ORA — Absolute
0E — ASL — Absolute
0F — NOP
10 — BPL
11 — ORA — (Indirect), Y
12 — NOP
13 — NOP
14 — NOP
15 — ORA — Zero Page, X
16 — ASL — Zero Page, X
17 — NOP
18 — CLC
19 — ORA — Absolute, Y
1A — NOP
1B — NOP
1C — NOP
1D — ORA — Absolute, X
1E — ASL — Absolute, X
1F — NOP
20 — JSR
21 — AND — (Indirect, X)
22 — NOP
23 — NOP
24 — BIT — Zero Page
25 — AND — Zero Page
26 — ROL — Zero Page
27 — NOP
28 — PLP
29 — AND — Immediate
2A — ROL — Accumulator
2B — NOP
2C — BIT — Absolute
2D — AND — Absolute
2E — ROL — Absolute
2F — NOP
30 — BMI
31 — AND — (Indirect), Y
32 — NOP
33 — NOP
34 — NOP
35 — AND — Zero Page, X
36 — ROL — Zero Page, X
37 — NOP
38 — SEC
39 — AND — Absolute, Y
3A — NOP
3B — NOP
3C — NOP
3D — AND — Absolute, X
3E — ROL — Absolute, X
3F — NOP
40 — RTI
41 — EOR — (Indirect, X)
42 — NOP
43 — NOP
44 — NOP
45 — EOR — Zero Page
46 — LSR — Zero Page
47 — NOP
48 — PHA
49 — EOR — Immediate
4A — LSR — Accumulator
4B — NOP
4C — JMP — Absolute
4D — EOR — Absolute
4E — LSR — Absolute
4F — NOP
50 — BVC
51 — EOR (Indirect), Y
52 — NOP
53 — NOP
54 — NOP
55 — EOR — Zero Page, X
56 — LSR — Zero Page, X
57 — NOP
58 — CLI
59 — EOR — Absolute, Y
5A — NOP
5B — NOP
5C — NOP
5D — EOR — Absolute, X
5E — LSR — Absolute, X
5F — NOP
60 — RTS
61 — ADC — (Indirect, X)
62 — NOP
63 — NOP
64 — NOP
65 — ADC — Zero Page
66 — ROR — Zero Page
67 — NOP
68 — PLA
69 — ADC — Immediate
6A — ROR — Accumulator
6B — NOP
6C — JMP — (Indirect)
6D — ADC — Absolute
6E — ROR — Absolute
6F — NOP
70 — BVS
71 — ADC — (Indirect), Y
72 — NOP
73 — NOP
74 — NOP
75 — ADC — Zero Page, X
76 — ROR — Zero Page, X
77 — NOP
78 — SEI
79 — ADC — Absolute, Y
7A — NOP
7B — NOP
7C — NOP
7D — ADC — Absolute, X
7E — ROR — Absolute, X
7F — NOP
80 — NOP
81 — STA — (Indirect, X)
82 — NOP
83 — NOP
84 — STY — Zero Page
85 — STA — Zero Page
86 — STX — Zero Page
87 — NOP
88 — DEY
89 — NOP
8A — TXA
8B — NOP
8C — STY — Absolute

十六進位數操作碼

8D – STA – Absolute
8E – STX – Absolute
8F – NOP
90 – BCC
91 – STA – (Indirect), Y
92 – NOP
93 – NOP
94 – STY – Zero Page, X
95 – STA – Zero Page, X
96 – STX – Zero Page, Y
97 – NOP
98 – TYA
99 – STA – Absolute, Y
9A – TXS
9B – NOP
9C – NOP
9D – STA – Absolute, X
9E – NOP
9F – NOP
A0 – LDY – Immediate
A1 – LDA – (Indirect, X)
A2 – LDX – Immediate
A3 – NOP
A4 – LDY – Zero Page
A5 – LDA – Zero Page
A6 – LDX – Zero Page
A7 – NOP
A8 – TAY
A9 – LDA – Immediate
AA – TAX
AB – NOP
AC – LDY – Absolute
AD – Absolute
AE – LDX – Absolute
AF – NOP
B0 – BCS
B1 – LDA – (Indirect), Y
B2 – NOP
B3 – NOP
B4 – LDY – Zero Page, X
B5 – LDA – Zero Page, X
B6 – LDX – Zero Page, Y
B7 – NOP
B8 – CLV
B9 – LDA – Absolute, Y
BA – TSX
BB – NOP
BC – LDY – Absolute, X
BD – LDA – Absolute, X
BE – LDX – Absolute, Y
BF – NOP
C0 – CPY – Immediate
C1 – CMP – (Indirect, X)
C2 – NOP
C3 – NOP
C4 – CPY – Zero Page
C5 – CMP – Zero Page
C6 – DEC – Zero Page
C7 – NOP
C8 – INY
C9 – CMP – Immediate
CA – DEX
CB – NOP
CC – CPY – Absolute
CD – CMP – Absolute
CE – DEC – Absolute
CF – NOP
D0 – BNE
D1 – CMP – (Indirect), Y
D2 – NOP
D3 – NOP
D4 – NOP
D5 – CMP – Zero Page, X
D6 – DEC – Zero Page, X
D7 – NOP
D8 – CLD
D9 – CMP – Absolute, Y
DA – NOP
DB – NOP
DC – NOP
DD – CMP – Absolute, X
DE – DEC – Absolute, X
DF – NOP
E0 – CPX – Immediate
E1 – SBC – (Indirect, X)
E2 – NOP
E3 – NOP
E4 – CPX – Zero Page
E5 – SBC – Zero Page
E6 – INC – Zero Page
E7 – NOP
E8 – INX
E9 – SBC – Immediate
EA – NOP
EB – NOP
EC – CPX – Absolute
ED – SBC – Absolute
EE – INC – Absolute
EF – NOP
F0 – BEQ
F1 – SBC – (Indirect), Y
F2 – NOP
F3 – NOP
F4 – NOP
F5 – SBC – Zero Page, X
F6 – INC – Zero Page, X
F7 – NOP
F8 – SED
F9 – SBC – Absolute, Y
FA – NOP
FB – NOP
FC – NOP
FD – SBC – Absolute, X
FE – INC – Absolute, X
FF – NOP

果粉藏書

内部交流
G16/4APPLE组合语言
(中3—5/46)
D00180

51